INTEGRATING DECISION PROCEDURES FOR TEMPORAL VERIFICATION

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

By
Nikolaj S. Bjørner
November, 1998

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Zohar Manna
(Principal Adviser)

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

David L. Dill

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Domenico Cantone

Approved for the University Committee on Graduate Studies:

To Bodil and Marianne

Acknowledgements

Zohar Manna, my advisor, has not only provided financial support during my time as a Ph.D. student, but also been both a wise and joyful advisor. Should I knock on his door during some odd hour there is always a loud, clear, and encouraging "come in". He permanently shows himself in a good mood and tolerates my sharp tongue with the greatest ease. When I felt progress was less evident he was always ready to encourage me.

Members of the STeP group have over time been the richest source of lively discussions, including Tomás E. Uribe, Henny Sipma, Michael Colón, Bernd Finkbeiner, Mark Pichora, Uri Lerner, Anca Browne, Luca de Alfaro, Jeff Kamerer, Arjun Kapur, Anuchit Anuchitanukul, and Eddie Chang. Tomás has always been in an arms-reach away for the last four years and been able to tolerate my questionable jokes while participating on several experimental jurneys, one of these an Intel-sponsored trip to Israel for a month in 1995 conveniently scheduled during all the Jewish holidays so that we could take excursions to Golan and Cairo. A later more comprehensive STeP excursion to Israel in the spring of 1998 offered even more adventure including a historic bath in the Dead Sea.

While I have always regarded Grigory Mints as a real Mensch, his patience in introducing me to the wonders of logic and proof theory has been impressive. I also feel lucky to have passed the paths of Sol Feferman and Carsten Thomassen. As real mathematicians they are perhaps the best role models a student can hope for in showing good taste and high quality in science. It was a pleasure to collaborate closely with Mark Stickel, and have insightsful discussions with David Cyrluk, Amir Pnueli, Calogero Zarba, Natarajan Shankar, Saddek Bensalem, Vaughan Pratt, Amer Diwan, Orna Grümberg, Harish Devarajan, and Shankar Govindaraju. Support from John Reppy, Dave McQueen, and Lal George on SML/NJ has been invaluable in implementing STeP. I also thank my reading commitee David Dill and Domenico Cantone for agreeing to read this thesis, as well as Tomás E. Uribe, Shmuel Katz, Clark Barrett, Mark Stickel, and Richard Waldinger for their comments on earlier drafts.

Thanks to my parents Kari and Dines for bringing me to the world a stone's throw from where these lines are written, and my darling sister Charlotte. Most importantly I thank my wife Bodil for her patience and understanding when I promised or did not promise that I would graduate next year, and my daughter Marianne for being such a cutie pie and øjensten.

Financial acknowledgements

The thesis was supported in part by the National Science Foundation under grants CCR-95-27927 and CCR-98-04100, the Defense Advanced Research Projects Agency under NASA grant NAG2-892, a gift from Intel, ARO under grants DAAH04-95-1-0317, DAAH04-96-1-0122 and DAAG55-98-1-0471, ARO under MURI grant DAAH04-96-1-0341, and by Army contract DABT63-96-C-0096 (DARPA).

Contents

List of Tables

List of Figures

Chapter 1

STeP and decision procedures

This introductory chapter gives some background on STeP and temporal verification. While the thesis is about developing and integrating decision procedures we will here briefly give the background for their applications in temporal verification.

1.1 STeP

The Stanford Temporal Prover (STeP) is a system for computer-aided formal verification of reactive, real-time and hybrid systems based on their temporal specifications, expressed in linear-time temporal logic (LTL). STeP integrates model checking and deductive methods to allow the verification of a broad class of systems, including parameterized (N-component) circuit designs, parameterized (N-process) programs, and programs with infinite data domains.

Figure 1.1 presents an outline of the STeP system. The main inputs are a reactive system and a property to be proven for it, expressed as a temporal logic formula. The system can be a hardware or software description and may include real-time and hybrid components. Verification is performed by model checking or deductive means, or a combination of the two.

1.1.1 Transition systems

Our computational model for reactive systems is that of a *transition system*,

$$\mathcal{S} = \langle \mathcal{V}, \Theta, \mathcal{T}, \mathcal{J}, \mathcal{C} \rangle,$$

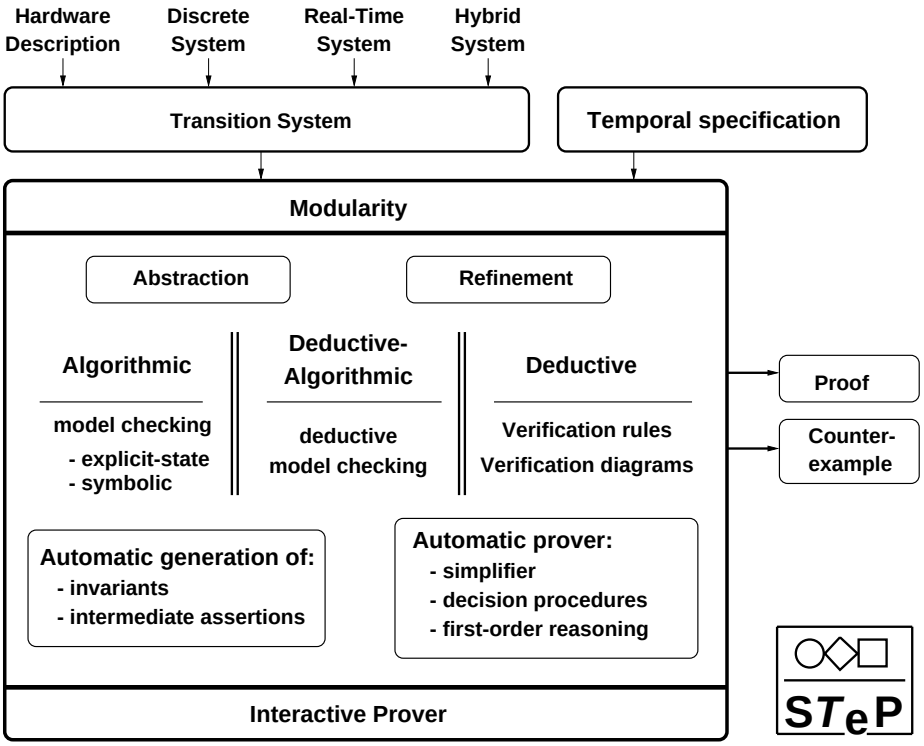


Figure 1.1: An outline of the STeP system

where $\mathcal{V}$ is a finite set of system variables, Θ is a formula characterizing the set of initial states, $\mathcal{T}$ is a finite set of transitions, $\mathcal{J} \subseteq \mathcal{T}$ is a set of *just* transitions, and $\mathcal{C} \subseteq \mathcal{T}$ is a set of *compassionate* transitions. The vocabulary $\mathcal{V}$ contains data variables, control variables and auxiliary variables. The set of *states* over $\mathcal{V}$ is denoted by Σ, where each state is an assignment of values to the variables in $\mathcal{V}$. The initial condition Θ is expressed as a first-order assertion. A transition τ maps each state $s \in \Sigma$ into a (possibly empty) set of τ-successors, $\tau(s) \subseteq \Sigma$. It is defined by an assertion $\rho_\tau(\overline{x}, \overline{x}')$, called the *transition relation*, which relates the values $\overline{x}$ of the variables in state s and the values $\overline{x}'$ in a successor state $s' \in \tau(s)$. We require that $\mathcal{T}$ contain a transition τ_I, called the *idling transition*, such that $\tau(s) = \{s\}$ for every state s. A transition τ is *enabled* on state s if $\tau(s) \neq \emptyset$.

A computation of a system $\mathcal{S}$ is an infinite sequence of states $s_0, s_1, s_2, \ldots$, such that (1) s_0 is an initial state satisfying Θ, (2) for every $i \geq 0$ there is a transition $\tau \in \mathcal{T}$ satisfying $s_{i+1} \in \tau(s_i)$, (3) for each $\tau \in \mathcal{J}$, if τ is enabled on states $s_i, s_{i+1}, s_{i+2}, \ldots$, then at some $j \geq i$, $\rho_\tau(s_j, s_{j+1})$ holds (in automata theory this is known as Büchi acceptance), (4) for

each $\tau \in \mathcal{C}$, if τ is enabled on infinitely many states, then τ is taken also on infinitely many states (in automata theory this is known as Streett acceptance).

Besides supporting input in the format of raw transition systems (actually elaborated with modularity [FMS98]), STeP also facilitates the representation of systems using a simple programming language (SPL [MP95, MAB+94]) with concurrency primitives. SPL statements are translated into transitions in a straightforward manner. For example, the assignment statement

$$\ell_0 : x := y + 1;\ \ell_1 :$$

assigns $y + 1$ to x when control resides at location ℓ_0, and subsequently moves control to ℓ_1. Control *labels* are encoded using control *counters* such that the property being at ℓ_0 is translated to $\pi = 0$ and being at ℓ_1 is translated to $\pi = 1$. One of the advantages of using counters is that they offer built-in detection of conflicting locations. In summary, the SPL compiler generates the transition τ with transition relation

$$\rho_\tau(\pi, x, y, \pi', x', y') : \quad \pi = 0 \ \wedge\ x' = y + 1 \ \wedge\ y' = y \ \wedge\ \pi' = 1 \ .$$

1.1.2 Linear-time temporal logic

The primary specification language used by STeP is first-order logic enhanced by temporal connectives [Pnu77].

A *temporal formula* is constructed from state formulas (called *assertions*), which are formulas from the first-order assertion language. To state formulas we apply boolean connectives (such as $\vee$, $\neg$), quantifiers ($\forall$, $\exists$) and temporal operators. The temporal operators used in this paper are future operators $\Box$ (*always in the future*), $\mathcal{W}$ (*waiting-for, unless*), $\bigcirc$ (*next*) and their past counterparts $\boxminus$ (*always in the past*), $\mathcal{B}$ (*back-to*) and $\ominus$ (*previously*).

A *model* for a temporal formula φ is an infinite sequence of states $\sigma : s_0, s_1, s_2, \ldots$, where each state s_j provides an interpretation for the variables occurring in φ. A temporal formula φ is $\mathcal{S}$-*valid*, written $\mathcal{S} \models \varphi$, if φ is satisfied on each computation σ of $\mathcal{S}$. This is written $\langle \sigma, 0 \rangle \models \varphi$. We define this relation below for the limited vocabulary used in this

thesis.

$$\langle \sigma, j \rangle \vDash \varphi \iff s_j \vDash \varphi \qquad \text{if } \varphi \text{ is first-order}$$

That is, φ is evaluated locally, using the interpretation in s_j

$$\langle \sigma, j \rangle \vDash \Box \varphi \iff \forall j' : j' \geq j \cdot \langle \sigma, j' \rangle \vDash \varphi$$

$$\langle \sigma, j \rangle \vDash \Diamond \varphi \iff \exists j' : j' \geq j \cdot \langle \sigma, j' \rangle \vDash \varphi$$

Furthermore we write $p \Rightarrow q$ as a shorthand for $\Box(p \rightarrow q)$.

1.2 Temporal verification

As Figure 1.1 indicates, STeP aims to support three verification paradigms: deductive, algorithmic, and a hybrid approach: deductive-algorithmic.

1.2.1 Deductive verification

The deductive methods of STeP verify temporal properties of systems by means of verification rules and verification diagrams. *Verification rules* reduce temporal properties of systems to first-order verification conditions [MP95]. The most widely used verification rule is INV given in Figure 1.2. It reduces the verification of the invariant $\Box p$ to the first-order

	For assertion p,
B1.	$\Theta \rightarrow p$
B2.	$\{p\}\ \mathcal{T}\ \{p\}$
	$\mathcal{S} \vDash \Box p$

Figure 1.2: Basic invariance rule INV.

verification conditions in premises $B1$ and $B2$. The condition $B2$ is shorthand for

$$\bigwedge_{\tau \in \mathcal{T}} (p(\overline{x}) \ \wedge \ \rho_\tau(\overline{x}, \overline{x}') \ \rightarrow \ p(\overline{x}')) \quad .$$

Verification diagrams [MP94, BMS95, Sip98] provide a visual language for guiding, organizing, and displaying proofs. STeP features a diagram editor that takes a system, a specification, and a diagram and generates the appropriate verification conditions.

Case studies of mainly deductive verification of infinite state parameterized systems are reported in [MAB+94, BLM97, BMSU97, BMSU98] and on the web at

`http://www-step.stanford.edu/case-studies` .

1.2.2 Algorithmic verification

Model checking [CE81] using either state space enumeration or symbolic methods [McM93] (using BDDs) can be used to prove temporal properties of systems with finite state spaces. An enumerative model checking algorithm for finite state systems and LTL is described in [MP95]. Its implementation in STeP applies to some infinite state systems as well. Symbolic model checking algorithms for deciding the general validity of linear time propositional temporal formulas as well as properties over a reactive system are described in [Bjø98b].

1.2.3 Deductive-algorithmic verification

A much discussed topic these days is *combining model checking with deduction*. The golden promise of this integration research is in adding the expressiveness of deductive techniques to the efficiency of model checking.

One approach taken within STeP has been Deductive Model Checking [SUM96]. Here, the state space exploration is performed symbolically using formulas to represent states and using decision procedures to incrementally guide the state space exploration.

A more separated approach is to generate abstractions of systems first and then model check the abstraction. Decision procedures are used to generate abstractions that preserve as much information as possible from the infinite state system. First results on this can be found in [CU98], and independent work within PVS and SMV is found in [BLO98]. Alternatively, one can use a theorem prover to in fact perform the state space exploration [GS97].

In all cases the general approach is only as good as the efficiency and expressiveness of the decision procedures. On the other hand, present experience has been that the abstracted systems are very small and can be handled within a second by good model checkers.

Abstraction is treated in depth in [Uri98].

1.3 Generating and strengthening invariants

An important component in bootstrapping deductive, algorithmic, and deductive-algorithmic verification are utilities for generating auxiliary invariants.

Deductive verification: Invariants that have either been generated automatically, or established using the INV rule can be used as assumptions when proving other invariants

using INV. However, the rule INV may fail, even in the presence of auxiliary invariants, that is

$$B2: \quad I_{aux} \ \rightarrow \ \{p\}\ \tau\ \{p\}$$

is not valid for some transition τ, where p is the invariant to be established, and I_{aux} is the set of auxiliary invariants. In this case one can strengthen the invariant candidate p to

$$p \ := \ p \ \wedge \ \text{WP}(\tau, p)$$

and try again. The operator $\text{WP}(\tau, p)$ is shorthand for $\forall \overline{x}' \ . \ \rho_\tau(\overline{x}, \overline{x}') \ \rightarrow \ p(\overline{x}')$. In general, we seek the greatest fix-point of the operator:

$$\mathcal{B}(X) \ \stackrel{\text{def}}{=} \ I_{aux} \ \wedge \ p \ \wedge \ \text{WP}(\mathcal{T}, X) \ .$$

The greatest fix-point of $\mathcal{B}(X)$ is written as $\nu X \ . \ \mathcal{B}(X)$.

To illustrate how auxiliary invariants can be used in conjunction with invariant strengthening consider a simplified version of Lamport's solution to the mutual exclusion problem for two processes, formulated in SPL in Figure 1.3.

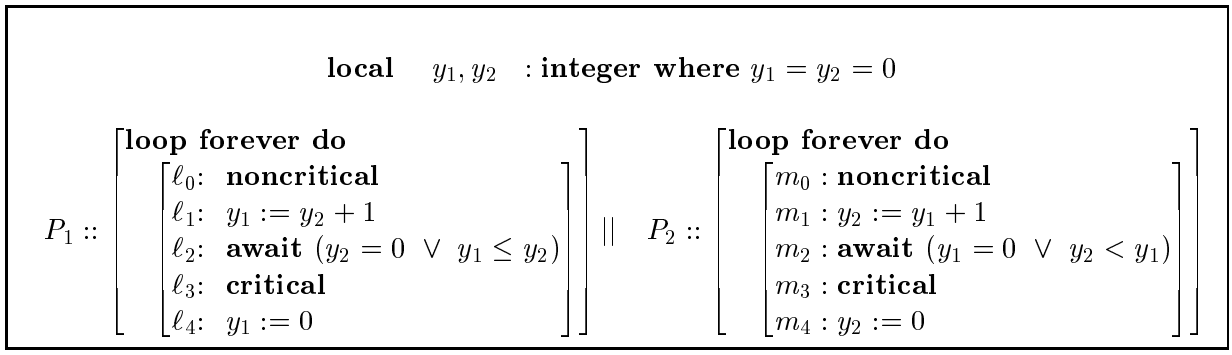


Figure 1.3: Program BAKERY (Program BAKERY for mutual exclusion)

STeP generates the auxiliary invariants $\Box(y_1 \geq 0)$ and $\Box(y_2 \geq 0)$ by propagating convex polyhedra. [1]

[1] In reality STeP generates several other invariants, so the strengthening done here becomes redundant, but let us pretend that STeP could only generate these weak assertions (which can also be inferred and checked by declaring y_1 and y_2 as natural numbers).

Backward propagation starts from an invariant candidate, in this case

$$\Box\,\neg(at\_\ell_3 \ \wedge \ at\_m_3),$$

which expresses mutual exclusion in the critical sections.

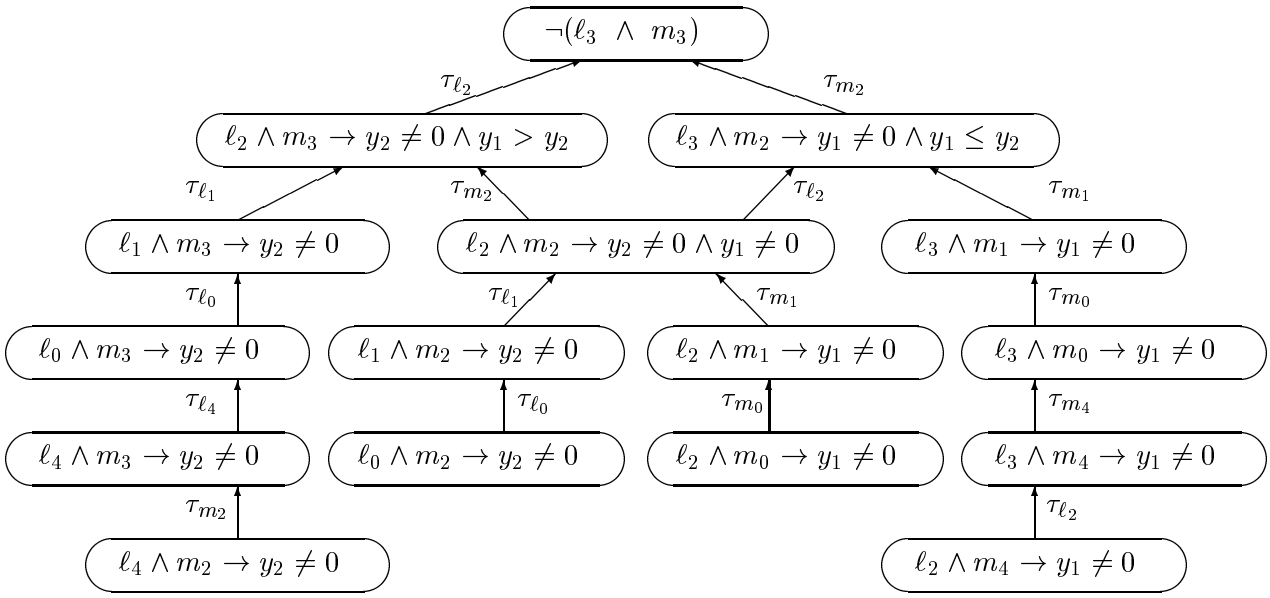


Figure 1.4: Backward propagation from $\neg(\ell_3 \ \wedge \ m_3)$

We compute the terms of the sequence

$$\underbrace{\mathrm{T}}_{\varphi_0} \leftarrow \underbrace{\mathcal{B}(\varphi_0)}_{\varphi_1} \leftarrow \underbrace{\mathcal{B}(\varphi_1)}_{\varphi_2} \leftarrow \cdots$$

until a limit is found. Applying $\mathcal{B}$ once generates $\varphi_1 : \neg(\ell_3 \ \wedge \ m_3)$. In the second iteration of $\mathcal{B}$ we calculate:

$$\begin{array}{rcl} \mathrm{WP}(\mathcal{T}, \varphi_1) & = & \bigwedge_{\tau \in \mathcal{T}} \mathrm{WP}(\tau, \varphi_1) \\ & = & \mathrm{WP}(\tau_{\ell_2}, \varphi_1) \ \wedge \ \mathrm{WP}(\tau_{m_2}, \varphi_1) \\ & = & (\ell_2 \wedge m_3 \to y_2 \neq 0 \wedge y_1 > y_2) \\ & & \wedge \ (\ell_3 \wedge m_2 \to y_1 \neq 0 \wedge y_1 \leq y_2). \end{array}$$

Continuing mechanically in this fashion we obtain the formulas shown in Figure 1.4. By calculating $\mathrm{WP}(\tau, \varphi)$, where τ labels an edge pointing to a φ-node, one obtains the assertion

labeling the source of the edge. The conjunction of the formulas is the greatest fix-point $\varphi_{\mathcal{B}}$ of $\mathcal{B}$. Finally, since $\Theta : \ \ell_0 \wedge m_0 \wedge y_1 = y_2 = 0$ implies $\varphi_{\mathcal{B}}$, we have indeed established mutual exclusion of the critical sections.

Algorithmic verification: Model checkers that use explicit state space exploration build the set of reachable states on the fly, and the use of auxiliary invariants has questionable advantages in this context. Symbolic model checking on the other hand, can be treated as a finite state instance of general assertional verification.

We will here abuse an older case study, a parameterized algorithm for mutual exclusion by Szymanski [SV94], in highlighting advantages of using auxiliary invariants in pruning symbolic model checking. The version of Szymanski's algorithm we examine is given in Figure 1.5.

$$
\begin{array}{lll}
\textbf{in} & N & : \textbf{integer} \ \ \textbf{where} \ N \geq 1 \\
\textbf{local} & a & : \textbf{array}\,[1..N]\ \textbf{of boolean} \ \ \textbf{where} \ \forall i : [1..N].\neg a[i] \\
 & s & : \textbf{array}\,[1..N]\ \textbf{of boolean} \ \ \textbf{where} \ \forall i : [1..N].\neg s[i] \\
 & w & : \textbf{array}\,[1..N]\ \textbf{of boolean} \ \ \textbf{where} \ \forall i : [1..N].\neg w[i]
\end{array}
$$

$$
\Big\|_{i=1}^{N} P[i] :: \left[\begin{array}{l}
\ell_0\text{: } \textbf{loop forever do} \\
\quad \left[\begin{array}{l}
\ell_1\text{: } \textbf{noncritical} \\
\ell_2\text{: } a[i] := \mathrm{T} \\
\ell_3\text{: } \textbf{await } \forall j : [1..N].\ \neg s[j] \\
\qquad \text{———— } \textit{doorway} \text{ ————} \\
\ell_4\text{: } (w[i], s[i]) := (\mathrm{T}, \mathrm{T}) \\
\qquad \text{———— } \textit{waiting room} \text{ ————} \\
\ell_5\text{: } \textbf{if } \exists j : [1..N].\ (a[j] \wedge \neg w[j]) \textbf{ then} \\
\quad \left[\begin{array}{l}
\ell_6\text{: } s[i] := \mathrm{F} \\
\ell_7\text{: } \textbf{await } \exists j : [1..N].\ (s[j] \wedge \neg w[j]) \\
\ell_8\text{: } s[i] := \mathrm{T}
\end{array}\right] \\
\qquad \text{———— } \textit{inner sanctum} \text{ ————} \\
\ell_9\text{: } w[i] := \mathrm{F} \\
\ell_{10}\text{: } \textbf{await } \forall j : [1..N].\ \neg w[j] \\
\ell_{11}\text{: } \textbf{await } \forall j : [1..(i-1)].\ \neg s[j] \\
\ell_{12}\text{: } \textbf{critical} \\
\ell_{13}\text{: } (s[i], a[i]) := (\mathrm{F}, \mathrm{F})
\end{array}\right]
\end{array}\right]
$$

Figure 1.5: Program SZY-A (Szymanski's algorithm: atomic version).

Some of the bottom-up invariants generated by STeP are:

$$\begin{aligned} \ell_{5,6,9..13}[i] &\leftrightarrow s[i] \\ \ell_{3..13}[i] &\leftrightarrow a[i] \\ \ell_{5..9}[i] &\leftrightarrow w[i] \end{aligned}$$

These are generated using assertion propagation noting that the variables $s[i]$, $a[i]$, and $w[i]$ are concurrently read, but exclusively written by process $P[i]$.

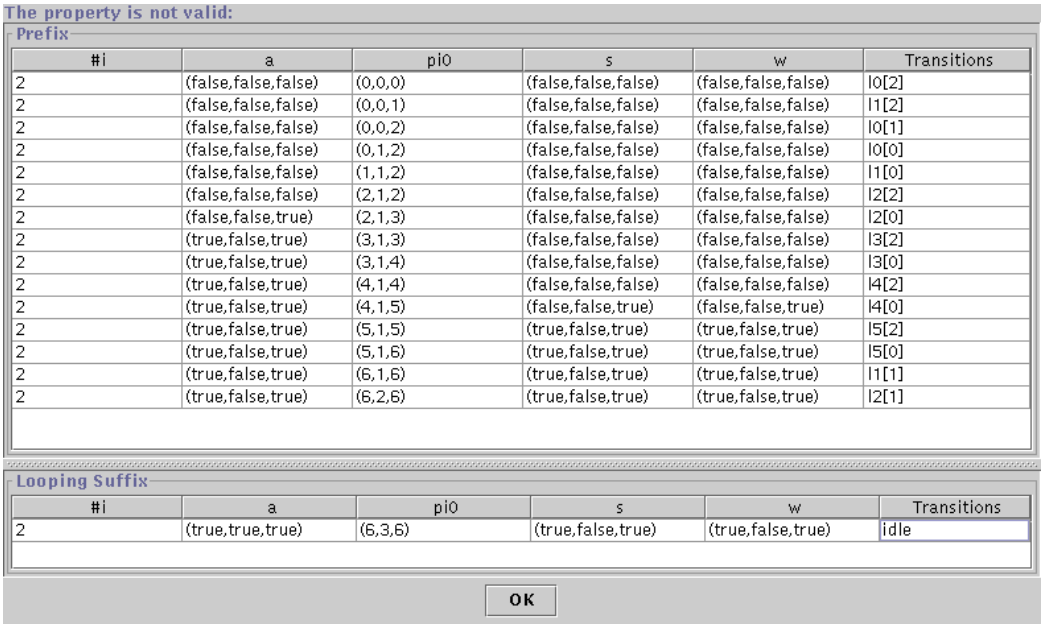
The main specifications of the algorithm include

$$\begin{aligned} &mux\text{:} && \ell_{12}[i] \wedge \ell_{12}[j] && \Rightarrow && i = j \\ &acc\text{:} && \ell_2[i] && \Rightarrow && \Diamond\, \ell_{12}[i] \end{aligned}$$

Finite instances of N can be checked directly using symbolic model checking because the system then becomes finite state. To check the invariant *mux* STeP computes $\nu X\ .\ \mathcal{B}(X)$ using OBDDs to maintain the intermediary assertions. It takes STeP a few seconds (3 on a SUN Ultra Sparc II) to check *mux* in the case with 3 processors. With 4 processors, checking takes about a minute and generates in excess of 1 million BDD nodes, with 5 processors, the checking takes about 30 minutes. The case with 6 processors takes about 2 hours to check. The situation is, on the other hand, much worse when applying the model-checking without the bottom-up invariants. With 3 processors the model checker now takes 3 minutes instead of 3 seconds. However, a direct computation of the reachable states takes 1 second with 3 processors thanks to the limited size of the example.

The *acc* property can also be established for the case of 3 processors. It takes the symbolic model checker 25 minutes to check this claim. Most of the time is spent on checking the fairness constraints imposed by the transition system.

Consider a version of Szymanski's algorithm without the conjunct $\neg w[j]$ in ℓ_5 and without statements ℓ_6 and ℓ_8. We can check that this program still satisfies mutual exclusion for 3 processors, but *acc* is violated because the program can deadlock, and the model checker reports a counter-example:


The property is not valid:

Prefix

#i	a	pi0	s	w	Transitions
2	(false,false,false)	(0,0,0)	(false,false,false)	(false,false,false)	l0[2]
2	(false,false,false)	(0,0,1)	(false,false,false)	(false,false,false)	l1[2]
2	(false,false,false)	(0,0,2)	(false,false,false)	(false,false,false)	l0[1]
2	(false,false,false)	(0,1,2)	(false,false,false)	(false,false,false)	l0[0]
2	(false,false,false)	(1,1,2)	(false,false,false)	(false,false,false)	l1[0]
2	(false,false,false)	(2,1,2)	(false,false,false)	(false,false,false)	l2[2]
2	(false,false,true)	(2,1,3)	(false,false,false)	(false,false,false)	l2[0]
2	(true,false,true)	(3,1,3)	(false,false,false)	(false,false,false)	l3[2]
2	(true,false,true)	(3,1,4)	(false,false,false)	(false,false,false)	l3[0]
2	(true,false,true)	(4,1,4)	(false,false,false)	(false,false,false)	l4[2]
2	(true,false,true)	(4,1,5)	(false,false,true)	(false,false,true)	l4[0]
2	(true,false,true)	(5,1,5)	(true,false,true)	(true,false,true)	l5[2]
2	(true,false,true)	(5,1,6)	(true,false,true)	(true,false,true)	l5[0]
2	(true,false,true)	(6,1,6)	(true,false,true)	(true,false,true)	l1[1]
2	(true,false,true)	(6,2,6)	(true,false,true)	(true,false,true)	l2[1]

Looping Suffix

#i	a	pi0	s	w	Transitions
2	(true,true,true)	(6,3,6)	(true,false,true)	(true,false,true)	idle

OK

It should be noted that in deductive verification of the algorithm, all verification conditions fall in the decidable class wS1S or M2L(str), and the Mona [BK95] tool can, in fact, be invoked from STeP to establish all verification conditions automatically (and in seconds) given the necessary strengthened invariants. Mutual exclusion for this algorithm can also be established algorithmically using symmetry reduction [SV94]. Finally, it is possible to compute the entire set of reachable states using regular automata used in Mona by propagating the transitions from the initial state (parameterized transition relations are modified such that independent actions can be taken simultaneously).

Deductive-Algorithmic verification: Auxiliary invariants can also be used to generate more precise abstractions as done in [CU98].

1.3.1 Methods for invariant generation

STeP contains utilities for propagating assertions based on the abstract syntax tree of SPL. These are called *local invariants*. Independent of SPL, STeP also contains utilities for generating invariants using techniques from linear algebra and linear programming. Theoretical extensions of these ideas can be found in [BBM95, BBM97], where abstraction domains and fix-point computations for general safety formulas are investigated. Several other recent developments for reactive systems are [BLS96, SDB96]; the notion of *reaffirmed* invariants can be found here. Approximation techniques with applications for real-time systems was developed in [WD95]. Reaffirmed invariants can also be used for the modular verification of real-time systems, as exploited in [BMSU97], and specializations to hybrid systems are studied in [MS98]. Recent use of partitioned BDDs for hardware invariants are described in [GDHH98, GD98]. Theoretical foundations for abstraction are well presented in [CC77, LGS$^+$95, DGG94, Dam96]. Invariant generation has naturally had a long story in

the analysis of sequential and functional programs [GW75, KM76] with roots in the analysis of Algol68.

1.4 Decision procedures

The integration of decision procedures has a long history dating back to [NO78, Sho79, BM79], but has enjoyed an exciting time recently [CLS96, BS96, BDL96, Det96, TR98] in both theoretical work and as used in verification systems. This, because decision procedures make common tasks easy, and (selected) hard tasks possible[2]. The concurrent research around Palo Alto has stimulated the development and identification of faster and more expressive decision procedures. High quality work around the ultimate verification system PVS has been an initial source of inspiration. The highly optimized SVC checker on the other hand has led the way in impressive benchmarks and set high standards in which large examples can be done within reasonable time using decision procedures. The driving force behind my involvement in the integration of decision procedures has been a desire to find well-tuned integrations of decision procedures for expressive theories, and widen the scope of decidable classes.

From a pragmatic point of view, decision procedures should ideally terminate quickly when the formula is not valid (or not in the scope of the supported theories), and not monopolize computing resources in proving valid formulas. As timing becomes critical, when thousands of calls are made to the decision procedures, low overhead is important for smaller examples; on the other hand, larger examples that are developed by careful manual modeling should also be handled whenever the used theories are in the scope of the decision procedures. For non-valid goals, feedback can be given in a variety of ways: as an assignment of rationals to the parameters of linear programming problems, for example. More often, however, a truth assignment to the atomic predicates in the goal may better communicate the source of the invalidity.

1.5 The rest of the thesis

A high-level framework for the integration of decision procedures is presented in Chapter 2. It surveys known approaches [Opp80a, Sho84] and ends up proposing a constraint-based version of Shostak's integration. On the other hand, we augment the Davis-Putnam procedure with rules for reasoning about first-order quantification. The ambition here is to blend first-order reasoning with the decision procedures that mainly work for quantifier-free formulas. The framework is intended to approach concrete problems in verification and the chapter does not provide deep new theoretical results. Although it extends Shostak's *algebraically solvable theories*, it relies on each decision procedure to provide what corresponds to a finite set of unifiers and therefore does not enjoy the full generality of the Nelson-Oppen framework. The rest of the thesis therefore examines theories that are central to temporal

[2]This is an adaption of a quote used to promote Perl.

verification to demonstrate how decision procedures for these theories fit into the proposed framework. An attempt is made to demonstrate that the framework does indeed allow the combination of an extensive set of theories. The approach, however, requires some insight to the workings of the individual decision procedures. Thus, the thesis presents:

Chapter 3: a new efficient combination of decision procedures based on congruence closure. A special feature is that it supports theories with self-referential (cyclic) data types,

Chapter 4: algorithms for integration of general special relations which go beyond the limitations of equality-based theory interfaces,

Chapter 5: cooperating decision procedures for linear and non-linear programming,

Chapter 6: algorithms for cyclic and acyclic recursive data types,

Chapter 7: bit-vector decision procedures, including non-fixed length bit-vectors,

Chapter 8: decision procedures for lists and queues.

A high-level overview of the proposed framework can be found in Figure 1.6.

Sections 2.3 and 2.4 are extracted from joint work with Mark Stickel and Tomás Uribe [BSU97], and Chapter 7 resulted from joint work with Mark Pichora [BP98]. In particular, Section 2.4 is due to Mark Stickel, and only close collaboration with Mark Pichora made Chapter 7 possible. For instance, Mark Pichora provided the necessary number theory to solve non-fixed size bit-vectors.

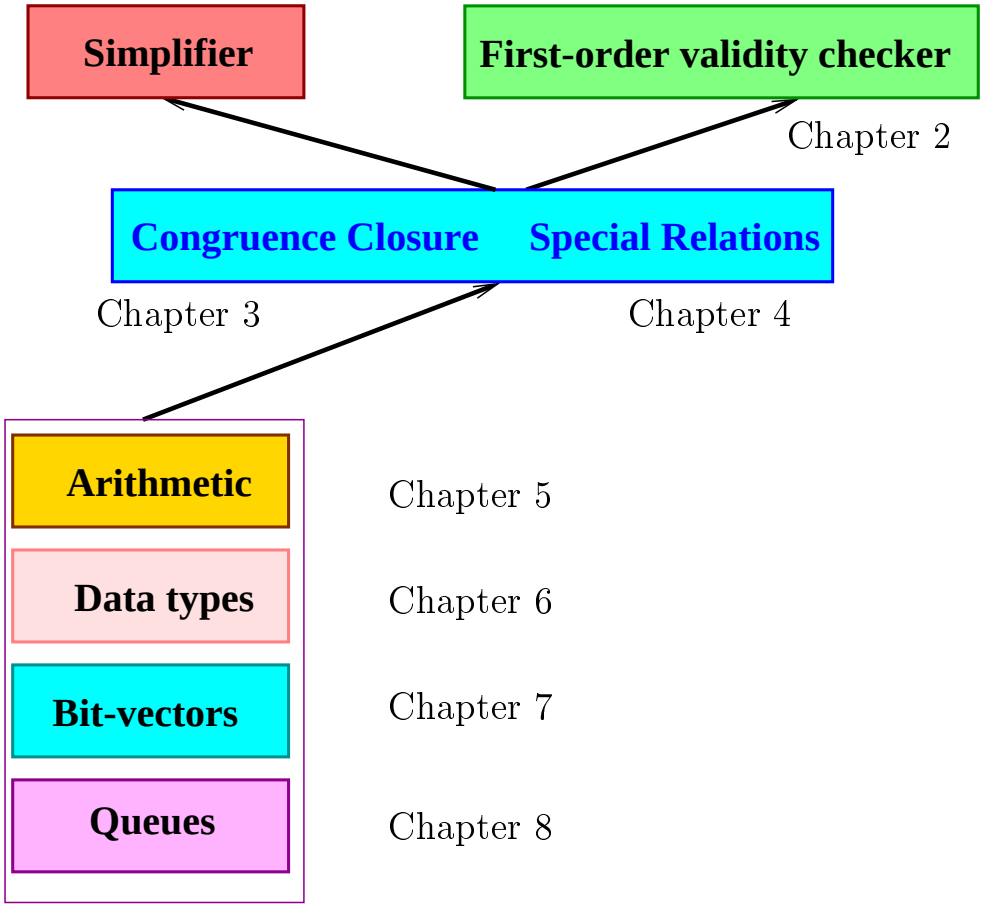


Figure 1.6: An overview of the integration of decision procedures

Chapter 2

Combining theories

In this chapter we will first review selected approaches in combining decision procedures, arriving at a constraint-based integration of solvers. Section 2.3 describes in a deductive style the proof-search procedure mixing first-order reasoning and decision procedures. We finish by describing the highlights of an implementation of the deductive component and explain where the decision procedures are coupled.

Figure 2.1 gives a rough overview of the proposed search paradigm.

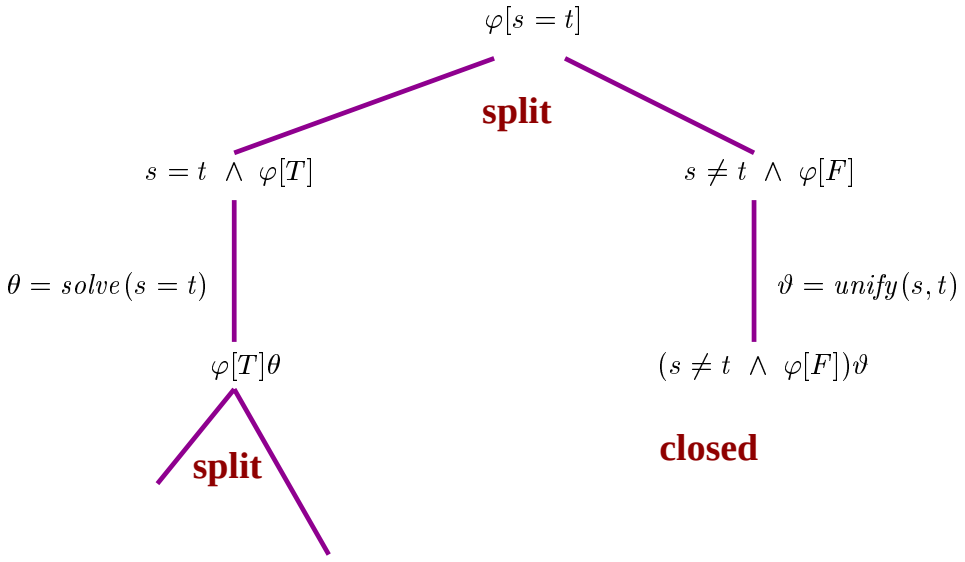


Figure 2.1: An overview of the refutation search

Suppose φ is the negated, skolemized version of some formula we wish to prove valid. The refutation search proceeds with a Davis-Putnam style case splitting. In the left branch

where an equality is asserted a congruence closure based integration of decision procedures is used to *solve* the asserted equality and reduce (rewrite) the resulting branch under that assumption. The right branch can be closed by possibly finding suitable instantiations of the skolem variables that unify terms s and t. As usual in tableau search the unifier ϑ must be applied to all branches in the search.

The rest of this chapter examines the appropriate tools and rules for realizing this verification approach.

2.1 Preliminaries

Formulas and expressions: STeP uses a first-order language with a rich and flexible syntax. Formulas are in nonclausal form, and boolean formulas can be nested inside arbitrary function symbols (for instance, p is under the function symbol f in $f(\mathit{if}\ p\ \mathit{then}\ t_1\ \mathit{else}\ t_2)$). An essential construct is *let-binding*, which explicitly represents structure sharing within an expression.

Therefore, our expressions will include first-order quantification, the usual set of boolean connectives and relations ($\vee, \wedge, \neg, \rightarrow, \leftrightarrow, \texttt{if-then-else}, \doteq$), and the construct $\texttt{let}\ x = e_1\ \texttt{in}\ e_2$ for a variable x and arbitrary expressions e_1 and e_2. The scope of x is e_2; occurrences of x in e_1 are free.

For a given formula $\mathcal{F}$, the *universal closure* of $\mathcal{F}$, written $\forall *.\mathcal{F}$, is the formula $\forall x_1..\forall x_n.\mathcal{F}$ where $\{x_1, \ldots, x_n\}$ are the free variables of $\mathcal{F}$. The *existential closure* of $\mathcal{F}$, written $\exists *.\mathcal{F}$, is defined similarly.

A *substitution* θ is a mapping $\theta : [x_1 \mapsto t_1, \ldots, x_n \mapsto t_n]$, where $x_1, \ldots, x_n$ are distinct variables and $t_1, \ldots, t_n$ are terms. For an expression e, $e\theta$ is the result of simultaneously replacing all free occurrences of x_i by t_i. Replacement is always *safe*, in that quantified variables are renamed to prevent capture, and bound variables are not replaced (see [MW93]). For substitutions θ and ρ, $\theta \cdot \rho$ is the substitution such that $x(\theta \cdot \rho) = (x\theta)\rho$. The substitution θ is *more general* than ρ if $\theta \cdot \gamma = \rho$ for some γ. The empty substitution is written as $[]$.

An *atom* is a formula with no boolean connectives; a *literal* is an atom or its negation. A *top-level conjunct* of a formula $\mathcal{F}$ is one of $\mathcal{F}_i$ if $\mathcal{F}$ is of the form $\mathcal{F}_1 \wedge \cdots \wedge \mathcal{F}_n$, and $\mathcal{F}$ otherwise. A *top-level literal* is a top-level conjunct that is a literal. We write $\mathcal{F}[e]$ for a formula with one or more occurrences of subexpression e, where e does not occur within the scope of a quantifier.

Sorts: STeP's object langauge uses sorts such as booleans, integers, rationals, reals, recursive data-types, records, function space, and queues. The symbols $\tau, S_1, \ldots, S_n$ range over sorts. We use $\mathcal{B}$ for booleans, $\mathcal{N}$ for naturals, $\mathcal{Z}$ for integers, and $\mathcal{R}$ for reals, in both the object and meta-language.

Polarity: We define the *polarity* of a subexpression in $\mathcal{F}$ in the usual way [MW93]: an occurrence of a subexpression e is *positive* (resp. *negative*) in $\mathcal{F}$ if it occurs within an even (resp. odd) number of negations, written as $\mathcal{F}[e]^+$ (resp. $\mathcal{F}[e]^-$). An occurrence has *both polarities*, written as $\mathcal{F}[e]^\pm$, if it appears under the $\leftrightarrow$ boolean connective or in the

if-clause of an `if-then-else` expression. If e has two occurrences in $\mathcal{F}$, one positive, and one negative, we can refer to both occurrences by $\mathcal{F}[e]^{\pm}$.

$\mathcal{F}[e]^{+}$, $\mathcal{F}[e]^{-}$ and $\mathcal{F}[e]^{\pm}$ respectively denote strictly positive, strictly negative and bipolar occurrences of e in $\mathcal{F}$.

Theories and decision procedures: Our goal is to decide general validity with respect to a background theory or combination of theories $\mathcal{T}$ (not necessarily complete or first-order axiomatizable). Following [BFP92], we define the following semantic properties of formulas:

Definition 2.1.1 *A closed sentence $\mathcal{F}$ is* $\mathcal{T}$-valid *if every model of $\mathcal{T}$ satisfies $\mathcal{F}$, and* $\mathcal{T}$-unsatisfiable *if no model of $\mathcal{T}$ satisfies $\mathcal{F}$.*

Definition 2.1.2 ($\mathcal{T}$-complementary) *A sentence $\mathcal{F}$ is* $\mathcal{T}$-complementary *if $\exists * .\mathcal{F}$ is $\mathcal{T}$-unsatisfiable.*

Definition 2.1.3 ($\mathcal{T}$-refuter) *θ is a* $\mathcal{T}$-refuter, *or* $\mathcal{T}$-refuting substitution, *for a sentence $\mathcal{F}$ if $\mathcal{F}\theta$ is $\mathcal{T}$-complementary.*

The last two notions are extended to sets of formulas by identifying a set with the conjunction of its elements. $\mathcal{T}$-complementary sets of literals in theory reasoning correspond to syntactically complementary pairs of literals in resolution—no instance is satisfiable in the theory.

A *decision procedure* for a theory $\mathcal{T}$ should always be able to identify the $\mathcal{T}$-complementarity of a set of quantifier-free literals.[1] However, if $\mathcal{T}$ is a combination of theories, each with its own decision procedure, we do not expect to obtain a combined decision procedure that is complete for the combined theory (i.e., not all $\mathcal{T}$-unsatisfiable sets will be identified). On some occasions, decision procedures will also be able to provide $\mathcal{T}$-refuting substitutions for a given set of literals.

In the rest of this paper, validity and satisfiability will always be understood relative to a theory $\mathcal{T}$, unless it is explicitly stated otherwise.

Function updates: If $f : A \to B$ is a function from domain A to range B, $a \in A$ and $b \in B$, then we write

$$f \dagger [a \mapsto b]$$

instead of

$$\lambda x \,.\, \textbf{if} \;\; x = a \;\textbf{then}\; \; b \;\textbf{else}\; \; f(x)$$

Operations on sets: We use $diag(S)$ as shorthand for $\{(x, x) \mid x \in S\}$. To restrict a function f to domain S we use $f \lceil S$.

[1] Note that decision procedures are not expected to reason about boolean formulas.

2.2 Integration of decision procedures

In the verification of reactive, real-time, hybrid systems, verification conditions generated from verification rules, abstraction, verification diagrams and dynamic flow analysis typically contain data-types that are used in the supplied systems. These data-types typically include integers, reals (for hybrid systems), arrays, recursive and co-recursive data-types, lists, queues, bit-vectors, etc.. It therefore becomes natural to provide customized support for the theories of each of the data-types. The most general and flexible approach is to support axiomatic presentation of theories in so-called theory libraries. As often axioms can be represented equivalently as rewrite rules (or be completed into a set of confluent rewrite rules) general support for rewriting is a way of giving efficient generic theory support. However, the general axiomatic approach, even with support from rewriting, cannot be expected to address decidability questions nor utilize specialized (efficient) data-structures when presented with an arbitrary theory. Even more optimized support can be provided for selected theories by providing decision procedures for these individually. While modular support for each data-type is desirable for a plug-and-prove combined decision procedure, some glue mechanism is required to achieve a complete integration of the provided decision procedures.

The issues involved in combining decision procedures have been studied decades ago starting with Nelson and Oppen as well as Shostak [NO79, Sho79, Opp80a, Sho84]. The approach taken in this thesis builds on and extends [Sho84].

The Nelson-Oppen approach forms today the basis for verification systems like ESC [Det96], and EVES [CKM$^+$91], and SDVS [LFMM92]. In theoretical work on the word problem the Nelson-Oppen approach has received attention in [TH96, BT97].

Shostak's approach in combining *algebraically solvable* theories on the other hand forms the basis for integration of decision procedures in systems like PVS [ORR$^+$96], SVC [BDL96], and STeP. Thanks to the analysis in [CLS96] it has received renewed attention, including noteworthy applications in deciding bit-vector constraints [CMR97, BP98, BDL98]. The requirement of algebraic solvability can give the impression that the approach is severely limited in comparison with the Nelson-Oppen method. It is, for instance, not always clear how non-equational constraints should be supported in conjunction with algebraic solvability. Support for cyclic data-types is also impossible if the congruence closure algorithm at the center of the theory integration requires well-founded substitutions.

Our ambition has thus been to demonstrate how the "blindingly fast" congruence closure based approach suggested by Shostak does in fact admit rather expressive generalizations. In providing a compositional solution we obtain increased expressiveness without losing basic efficiency for the simpler cases, such as reasoning about pure uninterpreted function symbols. We can also reason about linear arithmetic constraints while being able to also handle non-linear constraints.

2.2.1 Modular combination of theories and decision procedures

A modular approach for handling constraints over different sorts such as integers, bit-vectors, and recursive data-types, is to provide a separate constraint solver for each sort and then *glue* these together by propagating derived equalities. This approach suffices when equality is the only shared relation symbol of the theories. The dispatching of which decision procedure should be used is based on the *principal sort* of a constraint. For example the principal sort of an equality $s = t$ is the sort of s (which is the sort of t). If this happens to be a rational, then the appropriate decision procedure is one for rational arithmetic. While the sort-based assumption can be considered a real restriction as we cannot deal with theories sharing function symbols, it has not yet surfaced in our experience with program verification. However, it has been the subject of interesting theoretical work in [TR98]. Very general approaches in combining positive existential theories are discussed in [BS98]. In Section 2.2.2 and 2.2.3 we will review two ways of propagating equalities.

2.2.2 The Nelson-Oppen combination

Nelson and Oppen proposed a fairly general framework for combining procedures deciding satisfiability of quantifier-free sentences. An early presentation is given in [Opp80a], and a rigorous analysis is performed in [TH96]. We will borrow notation from the latter source whenever possible. Given theories $\mathcal{T}_1$ and $\mathcal{T}_2$ over disjoint signatures Σ_1 and Σ_2 and decision procedures Sat_i $i = 1, 2$ that establish satisfiability of quantifier free formulas using only function symbols from Σ_i the Nelson-Oppen combination provides a way to combine Sat_1 and Sat_2 to a procedure $Sat_{1\&2}$ that can establish satisfiability in $\mathcal{T} = \mathcal{T}_1 \cup \mathcal{T}_2$. While we obviously assume that $\mathcal{T}$ is consistent (otherwise the decision problem is trivial), a stronger condition, *stable-infiniteness*, is required for the combination $\mathcal{T}$ to have a simple presentation which is also complete.

Definition 2.2.1 (Stable-infiniteness [Opp80a]) *A consistent, quantifier free theory $\mathcal{T}$ with signature Σ is called* stably-infinite *whenever, for every quantifier-free Σ-formula φ, if $\{\varphi\} \cup \mathcal{T}$ is consistent, then there is an infinite model satisfying $\{\varphi\} \cup \mathcal{T}$.*

The combination procedure establishes satisfiability of φ by first introducing an adequate supply of variables such that there are no terms in φ with functions nested from Σ_1 and Σ_2. To separate boolean reasoning from the satisfiability procedures we then split φ to disjunctive normal form and from this point work with conjunctions of literals. A *Decomposition Phase* then separates a conjunction into two parts. One part contains terms involving function symbols from Σ_1, the other functions from Σ_2. They both contain all equalities and disequalities of the form $u \neq v$, $u = v$, where both u and v are variables. We allow a non-deterministic step to guess for each pair of variables $\langle u, v \rangle$ whether to add $u = v$ or $u \neq v$.

A final *Check Phase* invokes the procedures Sat_1 and Sat_2 independently on each separated conjunction.

1. **Variable Abstraction**
   For each sub-term $f(t_1, \ldots, g(\overline{s}), \ldots, t_n)$ in φ where $f \in \Sigma_1$ and $g \in \Sigma_2$, or vice versa, replace $g(\overline{s})$ by a fresh variable z and add the equality $z = g(\overline{s})$ to φ.

   If $s = t$ is a sub-formula of φ, and neither s nor t are variables we introduce a variable u and replace $s = t$ by $s = u \wedge t = u$.

   This process eliminates nested functions from different signatures at the expense of adding new equalities.

2. **Normal form conversion**
   φ is then converted into disjunctive normal form, and we guess a disjunct ψ (which is a conjunction of literals).

3. **Decomposition Phase**

   (a) From ψ form two conjunctions, ψ_1 and ψ_2, where ψ_i contains all conjuncts of ψ which are either equalities between variables, or contain function symbols from Σ_i, $i = 1, 2$.

   (b) Choose a partition P (i.e., $P = \{\{x_1, x_2\}, \{x_3\}, \{x_4, \ldots, x_{10}\}, \ldots\}$) of the variables $\overline{x}$ shared between ψ_1 and ψ_2. Each equivalence class should naturally only contain variables of the same type.

   (c) Simplify ψ_1 and ψ_2 by replacing each variable in P by an equivalence class representative $[x]_P$.

   (d) Use Δ to assert the disequalities

   $$\{[x]_P \neq [y]_P \mid \text{whenever } [x]_P \text{ and } [y]_P \text{ are different equivalence classes}\} \ .$$

4. **Check Phase**

   - Check satisfiability of $\psi_1 \wedge \Delta$ using Sat_1.
   - Check satisfiability of $\psi_2 \wedge \Delta$ using Sat_2.

The procedure returns *satisfiable* if there is a disjunct in the CNF of φ and a partition P of shared variables such that both $\psi_1 \wedge \Delta$ and $\psi_2 \wedge \Delta$ pass the check phase. A proof of soundness and completeness for this procedure is given in [TH96] when $\mathcal{T}$ is stably-infinite.

A few observations are worth pointing out in connection with this approach: (1) Craig's interpolation theorem tells us that we indeed only need to share equality constraints on shared variables, assuming equality is the only shared relation or function symbol, (2) stable infiniteness is essential for restricting the partition P to only shared variables; if stable infiniteness cannot be assumed, a partition of all terms and variables suffices to obtain completeness [Opp80a] (as we will establish in a sorted setting later), (3) the decomposition phase is required in case of *non-convex* theories.

Definition 2.2.2 (Convexity) *A set of constraints $\mathcal{C}$ is* convex *relative to disequalities $t_i \neq s_i$ $i = 1, \ldots, n$ if whenever each disequality $t_i \neq s_i$ is consistent with $\mathcal{C}$, then $\mathcal{C} \cup \{s_1 \neq t_1, \ldots, s_n \neq t_n\}$ is consistent.*

Similarly, a theory is convex if any conjunction of literals expressed over its language is convex relative to any set of disequalities.

Examples of convex theories include rationals under addition, equality with uninterpreted function symbols, and certain theories of S-expressions.

For convex theories we do not need to guess a partition P. Instead, the Nelson-Oppen combination suggests using each decision procedure Sat_i to incrementally check whether $\psi_i \wedge x \neq y$ is satisfiable for shared variables x, y. Consequently, if the decision procedures Sat_i for convex theories run in polynomial time on a conjunction of literals, the resulting combination will also run in polynomial time on inputs expressed as conjunctions of literals.

Shostak's solver-based combination optimizes the integration of decision procedures for convex theories that admit canonizers and solvers. We describe this next.

2.2.3 Shostak's combination

Shostak's method of combining decision procedures allows integrating decision procedures for theories, such as arrays, linear arithmetic over rationals, records, suitable data-types, simple set-theory and graphs inside Shostak's congruence closure algorithm [Sho84, CLS96, Mos88]. The method requires each theory $\mathcal{T}$ to provide (1) a *canonizer* (σ), which satisfies

1. $\sigma(s) = \sigma(t)$ whenever $\mathcal{T} \models s = t$. It follows that σ is idempotent.
2. If $\sigma(t) = f(t_1, \ldots, t_n)$ then $\sigma(t_i) = t_i$.

and (2) a *solver*, which rewrites an equation $s = t$ to either `false` (if it is unsatisfiable) or into an equivalent form $\exists V_{aux} \cdot \bigwedge_{i=1}^{n} x_i = t_i$, where

1. each x_i is an uninterpreted sub-term from s or t.
2. each t_i is canonized, i.e., $\sigma(t_i) = t_i$,
3. no x_i occurs in t_j,
4. no x_i is equal to an x_j, when $j \neq i$.
5. V_{aux} is the collection of auxiliary variables that occur in the t_j's but not in the original equation $s = t$.

In the case that a theory provides a canonizer and a computable solver it is said to be algebraically solvable.

A note on "variables": We shall use the term *skolem variable* to refer to variables that are obtained from skolemization of universal force quantifiers. Skolem variables can be instantiated by arbitrary terms to close the refutation search. Relative to a fixed theory

$\mathcal{T}$ we also use the term *variable* (without the qualification "skolem") to refer to sub-terms that are not interpreted in $\mathcal{T}$.

Using the solved form: The advantage of algebraically solvable theories is that we can write the solution as an idempotent substitution

$$\theta = [x_i \mapsto t_i \mid i = 1, \ldots, n] \ .$$

Explained in a simplified way the substitution can be used to decide verification conditions of the form $s_1 = t_1 \wedge s_2 = t_2 \rightarrow s_3 = t_3$ by extracting θ_1 from $s_1 = t_1$, extracting θ_2 from $(s_2 = t_2)\theta_1$ and check if $\sigma(s_3\theta_1\theta_2)$ is identical to $\sigma(t_3\theta_1\theta_2)$.

An implementation within Shostak's congruence closure algorithm allows to process equalities and disequalities in any order. The substitutions θ are applied immediately and stored in a union-find structure. The effect of applying θ is propagated via congruence closure on super-terms of the terms appearing in the domain of θ. With the terminology of rewriting theory the substitutions correspond to normalization with respect to ground completions [GNP+93]. In reasonable implementations the checks for unsatisfiability (violation of disequalities) happen on a call-by-need basis, that is, only when the terms involved in a disequality are made equal.

Shostak's congruence closure algorithm achieving this task was first published with subtle mistakes and without a rigorous correctness argument. It is probably no exaggeration that it remains mysterious even for experts in automated deduction, if not for Shostak himself today. What makes it attractive is that congruence closure here serves in dispatching decision procedures and combining them tightly.

2.2.3.1 Combining solvers

To combine theories over disjoint signatures (every function symbol is only interpreted in at most one theory) the solvers for each theory treat sub-terms headed by function symbols that are not interpreted in that theory as variables. Solvers for disjoint theories are then combined by applying them to a *set* of equations rather than a single equation until a fixpoint is reached. Requirement 1 is then no longer sufficient to guarantee termination. For instance, in the constraint

$$\mathtt{CAR}(z) \quad = \quad \mathtt{CAR}(x) + \mathtt{CDR}(y) \tag{2.1}$$

a solver for S-expressions treats the right-hand side as a variable because + is not interpreted in the theory of S-expressions. It could then produce the solution $\mathtt{CAR}(x)+\mathtt{CDR}(y) = \mathtt{CAR}(z)$; then a solver for linear arithmetic interprets +, but not CAR, so it chooses to return the original equation. In this setting a solution could be to change requirement 1 to

1. x_i is a variable from s whenever possible.

We cannot always require x_i to be a variable from the left-hand side as the following example suggests:

$$\underbrace{\texttt{CONS}(\texttt{NIL}, x)}_{s} = \underbrace{\texttt{CONS}(y, z)}_{t} \quad .$$

A solved form is

$$y = \texttt{NIL}, \quad x = z$$

but requires y which only occurs in t on the right-hand side. A clear disadvantage of this restriction is that by forcing the solved form to use primarily the left side of an equation, unnecessarily large expressions may be generated. For example, in solving

$$\texttt{CAAAR}(x) \quad = \quad y \tag{2.2}$$

we are required to return

$$x = \texttt{CONS}(\texttt{CONS}(\texttt{CONS}(y, z_1), z_2), z_3)$$

instead of just swapping the equality.

Fortunately, our combination here avoids the problems from (2.1) and (2.2) by allowing a partial interpretation of the selectors CAR and CDR when they are applied to variables. Then the constraint (2.1) can only be solved by the arithmetical constraint solver, because the principal sort of that equation is one of $\mathcal{N}$, $\mathcal{Q}$, $\mathcal{R}$, $\mathcal{C}$. The constraint (2.2) is simplify solved as $[\texttt{CAAAR}(x) \mapsto y]$.

2.2.3.2 Comparisons

In a very good sense one can regard solvers as unification algorithms and the solver/canonizer constraints as requirements on the solver to return most general unifiers. Shostak's integration of solvable theories is then in principle a heuristic optimization of (prominent) special cases where the Nelson-Oppen applies: to convex theories admitting solvers and canonizers. Shostak's integration is then (obviously sound and) complete in the same cases and for the same reasons as Nelson and Oppen's approach. Non-convex theories can in some cases still be supported by having the canonizers return compound expressions containing conditionals, but this may not always be the best heuristic approach. Furthermore, when constraints other than pure equalities are involved the naive use of Shostak's method lacks even more flexibility. This has led us to a constraint-based extension of the method to benefit from its advantages while enabling extended expressibility.

2.2.4 Constraint-based combination of solvers

To provide a more flexible framework, still benefiting from Shostak's combination of solvers, we use a notion of *constraint contexts*. Each context stores constraints that cannot be reduced to equalities over a particular sort. Hence, one context is allocated for the domain of integers, rationals and reals, another for recursive and co-recursive data-types, another

for bit-vectors etc.. The constraint contexts are then used to maintain constraints over the particular sorts. A context $\mathcal{C}$ over sort S can be updated by adding a constraint c whose principal sort belongs to the sort S. On the other hand, we require that in adding new constraints to contexts we will be able to extract all newly implied equalities in the form of a substitution in the same way as Shostak's *solve* routine. For example, if $\mathcal{C}$ contains the inequalities $x \leq y, y \leq z - 1$, then adding the constraint $c : z \leq x + 1$ results in the substitution $\theta : [x \mapsto y,\ z \mapsto y + 1]$ and the reduced constraint context $\mathcal{C}\theta$. Non-convex theories are supported via a *split* operation which takes a context $\mathcal{C}$ and splits it into a list of contexts and substitutions $(\mathcal{C}_1, \theta_1), \ldots, (\mathcal{C}_n, \theta_n)$. Informally, we read the list as a disjunction of possible simplification of $\mathcal{C}$.

In describing the requirements on the extended utilities we borrow notation and terminology from the constraint logic programming literature [JM94], as we also here deal with maintaining constraints. For the theories we will be studying assume that

- Equality is part of every theory. Hence, for every term s and t whose principal sort belongs to a given theory, $s = t$ is a legal constraint.
- Constraints are closed under negation: If c is a constraint, then $\neg c$ is a constraint too.

Although a set of constraints $\mathcal{C}$ is in practice maintained by specialized data-structures, we interpret them as suitable first-order formulas. In particular, `true` stands for the empty constraint context, and `false` for the unsatisfiable constraint context.

The theories and associated decision procedures described in more detail in chapters 5, 6, 7, and 8 are required to provide the operations *addConstraint*, *split*, and a canonizer σ. Recall that for each theory $\mathcal{T}$ is associated a language L (disjoint from languages over other theories, except for equality) and principal sort S (such as real, bit-vector, or queue). For the operations we require:

addConstraint: $\mathit{context} \times \mathit{constraint} \rightarrow \mathit{context} \times \mathit{substitution}$ As a generalization of the *solve* routine we use *addConstraint* to update a constraint context. In the case where $\mathcal{C}$ is the empty context, *addConstraint* and *solve* should coincide when presented with an equality. In this case *addConstraint* returns the empty context and a substitution.

For soundness we require that *addConstraint* is equivalence-preserving, i.e.,

$$\text{Let} \quad (\mathcal{C}', \theta') = \mathit{addConstraint}(\mathcal{C}, c). \qquad \text{Then} \qquad \mathcal{C} \ \wedge \ c \quad \leftrightarrow \quad \exists V_{aux} \ . \ \mathcal{C}' \ \wedge \ \theta' \ .$$

We add multiple constraints $c_1, \ldots, c_n$ using the notation

$$\mathit{addConstraints}(\mathcal{C}, \{c_1, \ldots, c_n\}) \ .$$

For flexibility we also admit substitutions that are not idempotent as long as they represent most general unifiers. For instance, a most general unifier for potentially cyclic terms can be expressed as a mapping on the term-graphs of terms s and t that are unified. While previous implementations of Shostak's congruence closure based

integration have been unable to handle such cyclic terms we will present an integration in Section 3 that does handle cyclic terms, and terminates on such unifiers as long as the non-well-founded solver does not need to introduce new terms.

σ : *context* $\times$ *term* $\rightarrow$ *context* $\times$ *term* The canonizer can also be made context-dependent which allows it to cause side-effects in contexts, such as accumulating splitter candidates. We write $\sigma_{\mathcal{C}}(t)$ as a shorthand for t', where $(\mathcal{C}', t') = \sigma(\mathcal{C}, t)$.

For instance, we will make the interpretation of data-type selectors such as `CAR` and `CDR` dependent on whether they take an argument labeled by a constructor (in this example `CONS`, `ATOM`, or `NIL`). When the canonizer processes a term of the form $\texttt{CAR}(x)$, where x is not a `CONS`, it returns $\texttt{CAR}(x)$, but stores x as a future splitter with the cases `NIL`, $\texttt{CONS}(y, z)$, or $\texttt{ATOM}(u)$.

split: *context* $\rightarrow$ (*context* $\times$ *substitution*)* $\cup \{true\}$ Primarily *split* allows to represent constraints over non-convex theories. Suppose that $x_1, \ldots, x_n$ are the variables (terms whose main function symbol is not in L) of sort S in $\mathcal{C}$, and let $s, t, s_1, t_1, \ldots, s_n, t_n$ below be terms whose variables of sort S are among $x_1, \ldots, x_n$.

From the arity of *split* it follows that either

$$split(\mathcal{C}) = true$$

or

$$split(\mathcal{C}) = \langle (\mathcal{C}_1, \theta_1), \ldots, (\mathcal{C}_n, \theta_n) \rangle$$

The first case represents the instance where no implicit equalities can be derived from $\mathcal{C}$. We require that:

$$\text{if} \quad \mathcal{C} \cup \mathcal{T} \models \bigvee_{i=1}^{n} s_i = t_i \quad \text{then} \quad \sigma_{\mathcal{C}}(s_i) = \sigma_{\mathcal{C}}(t_i) \text{ for some } i \ .$$

In other words, there is a model of $\mathcal{T}$ together with constraints $\mathcal{C}$ differentiating all terms over $x_1, \ldots, x_n$ unless the canonizer σ entails equality[2].

In the second case:

- For soundness we require that $\mathcal{C}$ imply the disjunction of the terms in *split*, i.e.,

$$\text{if} \quad \mathcal{C} \quad \text{then} \quad \bigvee_{i=1}^{n} \exists V_{aux} \ . \ \mathcal{C}_i \ \wedge \ \theta_i \ .$$

- For completeness we require *lazy equational completeness*. Let s and t be terms

[2]We need this implicit declaration of variables in $\mathcal{C}$ for the case of theories over finite domains.

built from declared variables,

$$\text{if} \quad \mathcal{T} \cup \mathcal{C} \models \bigvee_{i=1}^{m} s_i = t_i \quad \text{then} \quad \begin{array}{ll} & \sigma_{\mathcal{C}}(s_i) = \sigma_{\mathcal{C}}(t_i) \quad \text{for some } i \\ \text{or} & \mathcal{C} = \texttt{false} \\ \text{or} & \mathcal{T} \cup \mathcal{C}_j \models (\bigvee_{i=1}^{m} s_i = t_i)\theta_j \text{ for each } j = 1, \ldots, n \end{array}$$

- For termination we require there be a well-founded ordering $\prec$, such that $\mathcal{C}_i \prec \mathcal{C}$.

From soundness, completeness and termination it follows that *split* must provide a decision procedure for determining whether a conjunction of constraints $c_1, \ldots, c_n$ is satisfiable or not. This follows, as assume $c_1, \ldots, c_n$ is unsatisfiable, then for fresh variables x_{new}, y_{new} that do not appear in $c_1, \ldots, c_n$ we have

$$c_1 \wedge \ldots \wedge c_n \models x_{new} = y_{new} \ .$$

trivially as the assumptions are false. The three requirements imply that if we form $\mathcal{C}_0 = \texttt{true}$, and for $i = 1, \ldots, n$ and generate $(\mathcal{C}_i, \theta_i) = addConstraint(\mathcal{C}_{i-1}, (c_i)\theta_1 \cdots \theta_{i-1})$, then, either $\mathcal{C}_n = \texttt{false}$ or the result of applying *split* exhaustively to $\mathcal{C}_n$ results in the empty list (when interpreted as the empty disjunction this is `false`).

The availability of *split* gives us the freedom to require that if $\mathcal{C}$ implies some equality it is presented in a substitution θ after some sequence of splits. Implied equality constraints may thus be delayed at the discretion of the decision procedure. In place of *lazy equational completeness* one can desire *eager equational completeness*, which requires *addConstraint* to return in θ all equalities that are implied in conjunction with the new constraint. Thus, eager equational completeness states: If $c_1 \wedge \ldots \wedge c_n \models s = t$ and $\mathcal{C}_0 = \texttt{true}$, $(\mathcal{C}_i, \theta_i) = addConstraint(\mathcal{C}_i, (c_i)\theta_1 \cdots \theta_{i-1})$ for $i = 1, \ldots, n$, then either $\mathcal{C}_n$ is unsatisfiable or $\sigma(s\theta_1 \cdots \theta_n) = \sigma(t\theta_1 \cdots \theta_n)$. For the theory of linear arithmetic over the rationals we have an eager equational complete algorithm for maintaining arithmetical constraints.

2.2.4.1 Special relations

One of our interests will be to integrate decision procedures for theories that are essentially disjoint except for some sharing via *special relations* axioms of the form

$$x \prec_1 y \quad \rightarrow \quad f(x) \prec_2 f(y) \tag{2.3}$$

That is, assume we are given theories $\mathcal{T}_1$ and $\mathcal{T}_2$, over languages $\mathcal{L}_1$, $\mathcal{L}_2$ respectively, where $\prec_1 \in \mathcal{L}_1$ and $f, \prec_2 \in \mathcal{L}_2$ and $\mathcal{L}_1 \cap \mathcal{L}_2 = \{\doteq\}$ (the languages are disjoint, except for sharing equality). We now form the theory

$$\mathcal{T}_1 \cup \mathcal{T}_2 \cup \{x \prec_1 y \rightarrow f(x) \prec_2 f(y)\} \ .$$

In this case, Craig's interpolation theorem no longer suffices for combining disjoint

satisfiability procedures as the extra axiom combines the languages via more than equality reasoning.

We will treat special relations and mixed constraints in two methods (1) uninterpreted cases of special relations are handled as an extension of the congruence-closure algorithm (Chapter 4), (2) special relations involving arithmetical constraints are integrated with a linear arithmetic solver by adding extra interface utilities to the solver which allows other constraint solvers to access selected content (Chapters 5.1.4 and 6.2.6).

2.2.4.2 $\mathcal{T}$-refuting substitutions

To tackle instantiation of quantifiers our procedure will draw on utilities for finding $\mathcal{T}$-refuting substitutions. Only skolem variables are used in the domain of the instantiations.

instantiate : *context* $\rightarrow$ *substitution*$-$**set** In finding instantiations of quantified variables theories may provide an *instantiate* utility which given a context $\mathcal{C}$ returns a set of $\mathcal{T}$-refuting substitutions Θ, such that for $\theta \in \Theta$, $\mathcal{C}\theta$ is inconsistent.

In the case where $\mathcal{C}$ is a conjunction of pure uninterpreted equalities finding a $\mathcal{T}$-refuting substitution to $s \neq t$ is an NP-complete problem, known as the *rigid E-unification problem* (see 3.5). For extensions of equational theories with some special relation theories we show in Chapter 4 how to reduce the problem of finding $\mathcal{T}$-refuting substitutions to the rigid E-unification problem.

2.3 First-order refutation search: A calculus

Having presented generic requirements for integrating a class of decision procedures we will here continue with presenting the main framework in which boolean connectives and quantifiers are handled.

Integrating specialized decision procedures into general first-order theorem proving systems is a much-discussed problem with a long line of research [Plo72]. Much of this work has been carried out in the context of resolution, including theory resolution [Sti85], constrained resolution [Bür91], and the use of specialized unification [Fri91, BS93]. However, these methods usually make special demands on the decision procedures (computation of residues or complete sets of most general unifiers, identifying $\mathcal{T}$-unsatisfiable subsets, etc.). These requirements are not always satisfied by otherwise fast and efficient decision procedures. Furthermore, in a resolution setting they perform poorly on large formulas with a complex boolean structure.

Note that for some of the theories we consider, such as first-order logic with arithmetic, complete proof systems are impossible to obtain. However, our abstract procedure is complete for pure first-order logic (that is, the empty theory) and theories for which an appropriate version of Herbrand's theorem holds. This theoretical completeness claim holds for implementations that enumerate all possible substitutions. However, it does not hold for the much more effective selective generation of substitutions by unification and incomplete theory reasoning that we use in practice.

succeed	$\emptyset$	$\rightarrow$	refuted
reduce	$\{\mathit{false}\} \cup \mathcal{S}$	$\rightarrow$	$\mathcal{S}$
simplify	$\{\mathcal{F}\} \cup \mathcal{S}$	$\rightarrow$	$\{\mathit{SIMPLIFY}(\mathcal{F})\} \cup \mathcal{S}$
split	$\{\mathcal{F}[e]\} \cup \mathcal{S}$	$\rightarrow$	$\{e = d_i \ \wedge \ \mathcal{F} \mid d_i \in \mathbf{dom}(e)\} \cup \mathcal{S}$
instantiate	$\mathcal{S}$	$\rightarrow$	$\mathcal{S}\theta : \{\mathcal{F}\theta \mid \mathcal{F} \in \mathcal{S}\}$ for some substitution θ
skolemize$^+$	$\{\mathcal{F}[\forall x.\varphi]^+\} \cup \mathcal{S}$	$\rightarrow$	$\{\mathcal{F}[\varphi[y/x] \ \wedge \ \forall x.\varphi]^+\} \cup \mathcal{S}$
skolemize$^-$	$\{\mathcal{F}[\forall x.\varphi]^-\} \cup \mathcal{S}$	$\rightarrow$	$\{\mathcal{F}[\varphi[f_x(\overline{y})/x]]^-\} \cup \mathcal{S}$
skolemize$^\pm$	$\{\mathcal{F}[\forall x.\varphi]^\pm\} \cup \mathcal{S}$	$\rightarrow$	$\left\{ \begin{pmatrix} a_x(\overline{y}) \wedge \forall x.\varphi \\ \vee \\ \neg a_x(\overline{y}) \wedge \neg\forall x.\varphi \end{pmatrix} \wedge \ \mathcal{F}[a_x(\overline{y})]^\pm \right\} \cup \mathcal{S}$
let-eliminate	$\{\mathcal{F}\left[\begin{array}{l}\texttt{let } x = e_1 \\ \texttt{in } e_2\end{array}\right]\} \cup \mathcal{S}$	$\rightarrow$	$\{f_x(\overline{y}) = e_1 \ \wedge \ \mathcal{F}[e_2[f_x(\overline{y})/x]]\} \cup \mathcal{S}$

Figure 2.2: Rules for general $\mathcal{T}$-refuting procedure

Our procedure is an extension of the Davis-Putnam-Loveland-Logemann propositional satisfiability checker [DP60, DLL62]. It operates on formulas in nonclausal form, and is extended to consider quantifiers. The procedure is intended to preserve the original structure of the formula, including structure sharing using `let`- expressions, as much as possible. Case splitting, instantiation, skolemization and simplification can all be performed incrementally, in a uniform setting. We take advantage of instantiations suggested by decision procedures whenever available, but can also use "black-box" procedures that only provide yes/no answers.

For an arbitrary closed formula $\mathcal{G}$, *satisfiability-preserving skolemization* constructs a quantifier-free formula $\mathrm{Sk}(\mathcal{G})$ such that $\forall*.\mathrm{Sk}(\mathcal{G})$ is satisfiable iff $\mathcal{G}$ is satisfiable. $\mathcal{G}$ is valid iff $\neg\mathcal{G}$ is unsatisfiable, which is the case iff $\forall*.\mathrm{Sk}(\neg\mathcal{G})$ is unsatisfiable. This is the case if (but not only if) there is a ground-unsatisfiable instance $\mathrm{Sk}(\neg\mathcal{G})\theta$. Thus, the validity of a first-order formula can be established by finding a substitution for which a given quantifier-free formula is ground-unsatisfiable.

We now present a procedure in which skolemization, instantiation, quantifier duplication and the refutation search are all carried out within a unified framework. The procedure operates on a set $\mathcal{S}$ of formulas $\{\mathcal{F}_1, \ldots, \mathcal{F}_n\}$, where $\mathcal{S}$ is said to be satisfiable iff $\forall*.(\mathcal{F}_1 \vee \ldots \vee \mathcal{F}_n)$ is satisfiable. To finish a proof we need to show that all of the elements of $\mathcal{S}$ are, in fact, unsatisfiable, under a common instantiation. The abstract procedure proceeds by transforming the set $\mathcal{S}$, at each step applying one of the rules in Figure 2.2.

– **succeed**: This rule concludes the refutation search.

– **reduce**: *false* can be disregarded in the search for a satisfiable disjunct.

– **simplify**: $\mathcal{F}$ is simplified using equivalence preserving transformations, possibly to *false*, by the available decision procedures and simplification mechanisms (see Section 2.2). $SIMPLIFY(\mathcal{F})$ simplifies $\mathcal{F}$ with respect to its top-level literals, producing a formula $\mathcal{T}$-equivalent to $\mathcal{F}$. Minimal requirements for *SIMPLIFY* are:

- If $e = d_i$ is a top-level literal of the formula, then e occurs nowhere else in the simplified formula.
- If the top-level literals of the formula are recognized as $\mathcal{T}$-complementary, then the simplified formula is *false*.

– **split**: Subexpressions e taking values from a finite domain $\mathbf{dom}(e)$ can be analyzed according to the domain values. This includes boolean sub-formulas e, which are split with $e = \mathit{false}$ and $e = \mathit{true}$. In this case, the conjuncts added are $\neg e$ and e, respectively. Special cases of this rule are discussed in Section 2.4.

– **instantiate**: The substitution θ can instantiate free skolem variables in $\mathcal{S}$ by arbitrary (quantifier-free) terms.

– **skolemize**$^+$: y is a fresh variable.[3]

– **skolemize**$^-$: $\overline{y}$ is a tuple of all the free variables in $\forall x.\varphi$ and f_x is a fresh function symbol.

– **skolemize**$^\pm$: a_x is a fresh predicate symbol, and $\overline{y}$ is a tuple of all the free variables in $\forall x.\varphi$.

– **let-eliminate**: $\overline{y}$ is a tuple of all the free variables in e_1 and f_x is a fresh function symbol.

2.3.1 Main properties

We write $\mathcal{S} \rightarrow^* \mathcal{S}'$ if one or more rules transform the set $\mathcal{S}$ into the set $\mathcal{S}'$. We say that a rule *preserves satisfiability* when it transforms $\mathcal{S}$ to $\mathcal{S}'$, if:

$$\forall *. \bigvee_{\mathcal{F} \in \mathcal{S}} \mathcal{F} \text{ is } \mathcal{T}\text{-satisfiable} \quad \text{iff} \quad \forall *. \bigvee_{\mathcal{F} \in \mathcal{S}'} \mathcal{F} \text{ is } \mathcal{T}\text{-satisfiable.}$$

Lemma 2.3.1 *Except for* **instantiate**, *each rule in Section 2.3 preserves satisfiability when applied to any set $\mathcal{S}$. If the original set contains only closed formulas, and only these rules are applied, then* **instantiate** *preserves satisfiability as well.*

Proof:

[3]Similar skolemization rules apply to existential quantifiers, when $\exists x.\varphi$ has the opposite polarity.

We inspect each transformation rule. For an interpretation $\mathcal{I}$ (a model) a $*-$variant $\mathcal{I}^*$ is an interpretation that coincides with $\mathcal{I}$ except for the variables in $*$.

– **reduce**:$\forall *.\mathit{false} \vee \bigvee S$ is equivalent to $\forall *. \bigvee S$

– **split**: Every $\mathcal{I}^*$ corresponding to a satisfying model $\mathcal{I}$ for $\forall *. \{\mathcal{F}[e]\} \vee \bigvee \mathcal{S}$ satisfies $\{\mathcal{F}[e]\} \vee \bigvee S$. Clearly $\mathcal{I}^* \models e = d_i$ for some i, thus $\mathcal{I}^* \models \bigvee(\{e = d_i \ \wedge \ \mathcal{F} \mid d_i \in \mathbf{dom}(e)\} \cup S)$, and hence also $\mathcal{I}$ does.

– **simplify**: As the simplification is required to be equivalence preserving it trivially preserves satisfiability.

– **skolemize**$^+$: $\forall x.\varphi \ \rightarrow \varphi[y/x]$ so $\mathcal{F}[\forall x.\varphi \wedge \varphi[y/x]] \leftrightarrow \mathcal{F}[\forall x.\varphi]$.

– **skolemize**$^-$: Assume $\mathcal{I}^* \not\models \forall x.\varphi$, then there is a d depending only on the free variables in $\forall x.\varphi$, such that $\mathcal{I}^* \dagger [x \mapsto d] \not\models \varphi$. Augment $\mathcal{I}^*$ by skolem function $f_x(\overline{y})$ taking the free variables $\overline{y}$ in $\forall x.\varphi$ as argument such that whenever $\mathcal{I}^* \models \mathcal{F}[\forall x.\varphi]^-$ and $\mathcal{I}^* \not\models \forall x.\varphi$, then $[\![f_x(\overline{y})]\!] = d$ such that $\mathcal{I}^* \dagger [x \mapsto d] \not\models \varphi$.

On the other hand assume $\mathcal{I}^* \models \mathcal{F}[\varphi[f_x(\overline{y})/x]]$, then as $\forall x.\varphi \ \rightarrow \ \varphi[f_x(\overline{y})/x]$ the negative occurrence gives: $\mathcal{I}^* \models \mathcal{F}[\forall x \ . \ \varphi]$

– **let-eliminate**:

$$
\begin{array}{ll}
\mathcal{F}[\texttt{let } x = e_1 \texttt{ in } e_2] & \equiv \\
\exists x' \ . \ x' = e_1 \wedge \mathcal{F}[e_2[x'/x]] & \equiv \\
\exists x' \ . \ x' = e_1 \wedge \mathcal{F}[e_2[e_1/x]] & \equiv \\
(\exists x' \ . \ x' = e_1) \wedge \mathcal{F}[e_2[e_1/x]] & \text{is satisfiable if and only if} \\
f_x(\overline{y}) = e_1 \wedge \mathcal{F}[e_2[e_1/x]] & \equiv \\
f_x(\overline{y}) = e_1 \wedge \mathcal{F}[e_2[f_x(\overline{y})/x]] &
\end{array}
$$

– **skolemize**$^\pm$:

$$
\begin{array}{ll}
\mathcal{F}[\forall x \ . \ \varphi] & \equiv \\
\exists a \ . \ (a \leftrightarrow \forall x \ . \ \varphi) \wedge \mathcal{F}[a] & \equiv \\
\exists a \ . \ (a \leftrightarrow \forall x \ . \ \varphi) \wedge \mathcal{F}[\forall x \ . \ \varphi] & \equiv \\
(\exists a \ . \ (a \leftrightarrow \forall x \ . \ \varphi)) \wedge \mathcal{F}[\forall x \ . \ \varphi] & \text{is satisfiable if and only if} \\
(a_x(\overline{y}) \leftrightarrow \forall x \ . \ \varphi) \wedge \mathcal{F}[\forall x \ . \ \varphi] & \equiv \\
(a_x(\overline{y}) \leftrightarrow \forall x \ . \ \varphi) \wedge \mathcal{F}[a_x(\overline{y})] & \equiv \\
(a_x(\overline{y}) \wedge \forall x \ . \ \varphi \ \vee \ \neg a_x(\overline{y}) \wedge \neg \forall x \ . \ \varphi)) \wedge \mathcal{F}[a_x(\overline{y})] &
\end{array}
$$

In practice, we are only concerned with the "only if" direction of satisfiability preservation. This direction is always maintained by the **instantiate** rule, as well as the rule refinements we consider later on. If the old set is satisfiable only if the new one is, then we have:

Theorem 2.3.2 (Soundness) *For any closed formula $\mathcal{F}$, if $\{\neg\mathcal{F}\} \rightarrow^* \mathsf{refuted}$ then $\mathcal{F}$ is $\mathcal{T}$-valid.*

Rules that preserve satisfiability are *invertible*: if $\mathcal{S} \rightarrow \mathcal{S}'$ using an invertible rule, then $\mathcal{S} \rightarrow^* \mathsf{refuted}$ iff $\mathcal{S}' \rightarrow^* \mathsf{refuted}$. Lemma 2.3.1 tells us that all the rules in Section 2.3 are invertible. In particular, rules **reduce**, **skolemize**$^-$, and **let-eliminate** should be applied whenever possible, since they reduce the complexity of $\mathcal{S}$ and preserve satisfiability. Finally, we have:

Theorem 2.3.3 (First-order completeness) *Let $\mathcal{F}$ be a closed first-order formula. If $\mathcal{F}$ is (generally, or $\emptyset$-) valid then $\{\neg\mathcal{F}\} \rightarrow^* \mathsf{refuted}$.*

This follows, for example, from the completeness of the general matings procedure, given a suitable amplification of the formula [And81]. As in the case of resolution [Rob65], the completeness of most such procedures relies on Herbrand's theorem to guarantee that an appropriate finite ground instantiation always exists. Herbrand's theorem can be extended to account for certain classes of background theories [Fri91, BFP92, GNRS92]. Since practical implementations will sacrifice completeness by considering only instantiations with a limited amount of quantifier duplication, (see Section 2.3.2), we will not be concerned with ensuring that such an extended Herbrand theorem holds.

Theorem 2.3.4 (Ground decidability) *Let $\mathcal{F}$ be a closed formula where all occurrences of $\forall$ are strictly positive. If $\{\neg\mathcal{F}\} \rightarrow^* \{\mathcal{F}'\} \cup \mathcal{S}$ and $\mathcal{F}'$ is $\mathcal{T}$-consistent, then any $\mathcal{T}$-model for $\mathcal{F}'$ is also a model for $\neg\mathcal{F}$.*

Thus, if we can decide the $\mathcal{T}$-consistency of a formula $\mathcal{F}'$ obtained from the analysis of $\neg\mathcal{F}$, then we can conclude that $\mathcal{F}$ is not valid; a model for $\mathcal{F}'$ is a counterexample.

2.3.2 Equations, rewrites and limited quantifier duplication

To narrow the search, one can limit the number of quantifier duplications in rule **skolemize**$^+$. For most practical applications the quantifier need not be duplicated at all, using the following rule:

– **skolemize**$_0^+$**:** $\{\mathcal{F}[\forall x.\varphi]^+\} \cup \mathcal{S} \quad \rightarrow \quad \{\mathcal{F}[\varphi[y/x]\} \cup \mathcal{S}$.

In this case, rules **skolemize**$_0^+$ and **skolemize**$^\pm$ should take precedence over **split**, and the entire formula is fully skolemized before the search begins.

As a special case of quantifier duplication, conjuncts can be added whenever they are an immediate consequence of a universally quantified top-level literal.[4] A common case is that of equalities: if the formula $\forall * .(s = t)$ is known (variables renamed apart), one can add the rules:

– **rewrite:** $\{\mathcal{F}[e]\} \cup \mathcal{S} \quad \rightarrow \quad \{e = t\theta \ \wedge \ \mathcal{F}[e]\} \cup \mathcal{S} \qquad$ where $e = s\theta$.

– **narrow:** $\{\mathcal{F}[e]\} \cup \mathcal{S} \quad \rightarrow \quad (\{e = t \ \wedge \ \mathcal{F}[e]\} \cup \mathcal{S})\,\theta \quad$ where $e\theta = s\theta$.

In this way, equations that are not terminating or confluent can be applied step by step. A conditional rewrite rule, which rewrites e to e' under condition c, can be applied yielding $\{(c\theta \rightarrow e = e') \ \wedge \ \mathcal{F}[e]\} \cup \mathcal{S}$ or adding the equality $e = e'$ after ensuring that $c\theta$ holds under the assumption $\mathcal{F}[e]$.

2.3.3 Sequent calculus

The above presentation is analogous to proof in a Gentzen-style [Gen69] sequent calculus, where each transformation corresponds to a rule, and each element of the set $\mathcal{S}$ is a branch in the proof. To illustrate this, we show how well-founded (transfinite) induction and a *cut rule* can be added in very much the same way they are added to sequent-style calculi. (These rules are not part of our implementation, described in Section 2.4.)

induction	$\{\mathcal{F}[\forall x.\varphi]^-\} \cup \mathcal{S}$	$\rightarrow$	$\left\{\mathcal{F}\left[\forall x.\left(\begin{array}{l}\forall y.\,(y \prec x \rightarrow \varphi[y/x]) \\ \rightarrow \varphi\end{array}\right)\right]\right\} \cup \mathcal{S}$
cut	$\{\mathcal{F}\} \cup \mathcal{S}$	$\rightarrow$	$\{\mathcal{G} \wedge \mathcal{F},\ \neg\mathcal{G} \wedge \mathcal{F}\} \cup \mathcal{S}$ for an arbitrary formula $\mathcal{G}$.

In the **induction** rule, $\prec$ should be a well-founded order, and y a fresh variable.

A standard proof-theoretic analysis can demonstrate how to transform an arbitrary Gentzen-style derivation into a derivation of the calculus presented here, and vice-versa. Furthermore, a *cut-elimination* theorem holds for the first-order calculus presented here (without the induction rules [Min92]): derivations involving splits on non-atomic formulas can be converted into derivations using only splits on atomic formulas. Uses of rule **cut** can also be eliminated from the first-order (uninterpreted) calculus using a standard cut-elimination procedure.

2.4 Refutation search: Backtracking implementation

Following is a description of the nondeterministic refutation search procedure rewritten to suggest a practical implementation that uses depth-first search with backtracking. It assumes the formula has already been skolemized. When successful, $REFUTE\,(\mathcal{F}, [\,])$ returns

[4] Quantifier duplication in the ESC system [Det96] is in the form of such matching, limited by a heuristic bound.

a $\mathcal{T}$-refuting substitution for $\mathcal{F}$. Our inspiration for this approach is the Davis-Putnam-Loveland-Logemann (DPLL) propositional satisfiability procedure, which is effective and requires little memory.

$REFUTE(\mathcal{F}, \sigma_1) =$
  $\mathcal{F}' \leftarrow SIMPLIFY(\mathcal{F}\sigma_1)$
  if $\mathcal{F}' = false$ **then return** σ_1
  else do one of
    instantiate: $\theta \leftarrow$ a substitution
      return $REFUTE(\mathcal{F}', \sigma_1 \cdot \theta)$
    split: $e, \{d_1, \ldots, d_n\} \leftarrow$ an expression and possible values
      $\sigma_2 \leftarrow REFUTE(e = d_1 \wedge \mathcal{F}', \sigma_1)$
      $\sigma_3 \leftarrow REFUTE(e = d_2 \wedge \mathcal{F}', \sigma_2)$
      $\vdots$
      return $REFUTE(e = d_n \wedge \mathcal{F}', \sigma_n)$
    amplify: $\mathcal{G} \leftarrow$ an amplification formula
      return $REFUTE(\mathcal{G} \wedge \mathcal{F}', \sigma_1)$

The DPLL procedure is an instance of *REFUTE* when $\mathcal{F}$ is in clause form, *SIMPLIFY* implements unit resolution and subsumption, only the *split* operation is used, and e is an atomic formula that occurs in a nonunit clause. The added *instantiate* and *amplify* operations extend the substitution and formula respectively. The approach is reminiscent of the search for general matings [And81, Bib82, Iss90] except here paths are refuted by $\mathcal{T}$-complementary sets of literals instead of syntactically complementary pairs (cf. theory matings [Sti85]).

2.4.1 The basic operations

The instantiate operation: *instantiate* extends the current substitution σ_1 by a substitution θ chosen "don't know" nondeterministically with backtracking. Ideally, if $\mathcal{F}'$ is unsatisfiable, then θ should be a $\mathcal{T}$-refuting substitution for $\mathcal{F}'$. $\mathcal{T}$-refuters for the top-level literals of $\mathcal{F}'$ can sometimes be found and used as θ.[5] $\mathcal{T}$-refuters include substitutions that make literals complementary by ordinary unification; others may be proposed by the decision procedures (see Section 2.2); finally, others can be found using rigid E-unification (Section 3.5). In fact [DV96] have shown how to use a partial rigid E-unification procedure to provide a complete procedure for a tableau based first-order calculus. It can immediately be adapted to also work for our procedure. The only difference being that tableau rules split on logical connectives, whereas our procedure splits on atomic sub-formulas.

Saving substitutions σ_1 for which *REFUTE* fails enables elimination of redundant work due to duplicate substitutions.

[5] When a substitution known to be a $\mathcal{T}$-refuter of the top-level literals is chosen as θ, the succeeding call on *REFUTE* is guaranteed to succeed immediately and can be optimized away.

Trying to find a refutation using only the $\mathcal{T}$-refuters one knows about may seem overly optimistic, but it appears to work often enough to be a reasonable approach.

A second, less optimistic approach entails enumerating in advance possible values for the variables in the formula. The *instantiate* operation would then be used to generate the space of alternative substitutions. A good, but still incomplete way of finding possible values is to look at the positions of variables in the formula and then find terms that occur in complementary positions. For example, t is a possible value for x if x occurs as argument i of P and t as argument i of $\neg P$. The notion of complementary position can be extended in theory-specific ways, e.g., t and x are in complementary positions in $s \prec t$ and $x \prec y$ (see Section 4.1). Our current focus and examples use the first approach.

The split operation: *split* can select an atom e to split on with possible values *true* and *false* as in the DPLL procedure. When e or $\neg e$ is a top-level literal the procedure always prefers the implied unit-split. As an extension of the DPLL procedure, *split* can also select a nonconstant term e to split on with the elements of its finite domain $\{d_1, \ldots, d_n\}$ as values.

Good heuristic selection of what expression to split on can have a dramatic effect on the size of the search space. Unlike the DPLL procedure, we are using nonclausal, nonground formulas, but criteria similar to those used in the DPLL procedure [HV95] are useful, such as number of occurrences and the length of the shortest clause a literal would occur in if the formula were converted to clause form. Constraint satisfaction heuristics, such as preferring expressions with smaller domains to split on first, can also be used.

In the DPLL procedure, the selection of which atom to split and the order of values to try are "don't care" nondeterministic choices that affect the search space but not completeness. However, this selection can affect whether *REFUTE* succeeds or not. For *REFUTE*, we assume that $\mathcal{T}$-complementarity can be recognized, but not that $\mathcal{T}$-refuters can always be found. For example, the $\mathcal{T}$-complementarity of $P(2) \wedge \neg P(1+1)$ may be recognized without assuming that the decision procedures are also able to propose $\{x \mapsto 1\}$ as a $\mathcal{T}$-refuter of $P(2) \wedge \neg P(x+1)$. When *REFUTE* is applied to $P(2) \wedge Q(1) \wedge (\neg P(x+1) \vee \neg Q(x))$, some search orders would succeed, because $\{x \mapsto 1\}$ is discovered as a unifier for $Q(1) \wedge \neg Q(x)$ before attempting to refute $P(2) \wedge \neg P(x+1)$, while others would fail when the latter subproblem is encountered first. However, backtracking through alternative orders of splitting is combinatorially expensive, so we do not do it and accept this additional source of incompleteness.

The amplify operation: Davis [Dav81] defines *obvious inferences* as those that only require substitution for single instances of the formulas (i.e., no quantifier duplication is needed). The combination of *split* and *instantiate* is complete for obvious first-order inferences. It will also make some obvious $\mathcal{T}$-inferences, though not all. Even if $\mathcal{T}$ consists only of the theory of equality, the undecidability of simultaneous rigid E-unification [DV95] limits completeness of obvious $\mathcal{T}$-inference procedures.

Using only *split* and *instantiate* is our preferred approach. They are sufficient for several examples which we believe are typical problems for STeP. The search space is finite and often small. If quantifier duplication is allowed, the search space would be much larger (with limited duplication) or infinite (with unlimited duplication). The single-instance

restriction is a natural one that is readily understood by the user. The restriction is easily circumvented by the user's explicit inclusion of additional copies of the formulas (e.g, by manual application of **skolemize**$^+$).

Nevertheless, the *amplify* operation is allowed to do limited quantifier duplication, principally for the purpose of applying rewrites. Rather than duplicating a quantifier "in place", *amplify* is defined to add an amplification formula as a conjunct to the formula being refuted. The amplification formula may be any formula that can be soundly used in the refutation; it will typically be a fresh instance of a rewrite or premise (see Section 2.3.2).

2.4.2 Data structures

A detailed account of the data structures used to repesent terms and formulas is given in the next Chapter in connection with congruence closure. Similarly to [NO79] and [BDL96] we use a digraph representation of both terms and formulas, where the only boolean connective is **ite** (if-then-else expressions). In fact terms and formulas are not distinguished apart as we allow **ite** and quantification nested inside terms. We use the phrase *expression* to refer to terms and formulas. Hence, basic expressions are of the form:

$\mathbf{ite}(n_1, n_2, n_3)$	if-then-else with sub-terms indexed by n_1, n_2, n_3
$bind\ x : \tau.n$	τ is a sort and $bind \in \{\lambda, \int, \sum, \prod, \forall, \exists, \exists!\}$
$f(n_1, \ldots, n_k)$	(un)interpreted function application
true	
false	
x	variable

It is easy to translate let-expressions and standard boolean connectives to conditional normal form without increasing the size of the resulting term-graph. For example $\varphi \leftrightarrow \psi$ is translated into $\mathbf{ite}(\varphi, \psi, \neg\psi)$, where $\neg\psi$ is translated into $\mathbf{ite}(\psi, \mathbf{false}, \mathbf{true})$, and the same node is used to represent both occurrences of ψ. Implementing the translation using a hash-table makes maximal sharing automatic.

Potentially, the conversion into conditional normal form converts every atomic subformula into dual polarity position (in the head of an if-then-else test). The standard notions for *unit*-literals also break down without a notion of and-or formula representations. It is however possible to recover polarities and unit-literals based on the conditional normal form.

The SIMPLIFY operation: To propagate the effect of splitting, the congruence closure algorithm presented in the next Chapter propagates reductions of **ite** terms using canonizations of the form $\sigma(\mathbf{ite}(\mathbf{true}, b, c)) = b$ etc..

Polarity: Assume that node n is assigned positive polarity and that n points to $\mathbf{ite}(n_0, n_1, n_2)$. With the usual interpretation of **ite** both n_1 and n_2 occur with positive polarity, but n_0 has dual polarity. We can however assign n_0 positive polarity if the boolean expression associated with n implies a modified version of the boolean expression associated with n, where n_0 has been replaced by **true**.

This will be the case if n_2 implies n_1. A partial check for this can be obtained by checking if the set of positive sub-trees of n_2 form a positive *frontier* of the tree rooted at n_1. Hereby we define the relation specified via the predicate $\rightsquigarrow$:

$$n_2 \rightsquigarrow n_1 \stackrel{\text{def}}{=} \begin{array}{ll} & n_2 = \mathbf{false} \\ \text{or} & n_1 = \mathbf{true} \\ \text{or} & n_2 = \mathbf{ite}(a_1, a_2, a_3), \quad a_2 \rightsquigarrow n_1 \ \wedge \ a_3 \rightsquigarrow n_1 \\ \text{or} & n_1 = \mathbf{ite}(b_1, b_2, b_3), \quad n_2 \rightsquigarrow b_2 \ \wedge \ n_2 \rightsquigarrow b_3 \end{array}$$

This partial characterization has the advantage that it can be checked very quickly and preserves the polarities of standard formulas translated into conditional normal form. For example $\varphi \wedge \psi$ is represented as $n : \mathbf{ite}(n_0 : \varphi, n_1 : \psi, n_2 : \mathbf{false})$. If the polarity of n is $+$, then we should naturally assign polarity $+$ to n_1, and since $n_2 = \mathbf{false}$, we have $n_2 \rightsquigarrow n_1$, so n_0 should also have positive polarity.

A dual requirement holds for negative polarity.

Literal weight: A crucial heuristic that makes the Davis-Putnam procedure work efficiently is the ability to choose unit literals whenever possible to perform unit propagation.

We compute the weight of literals by assigning the top-most expression degree $\delta = 0$ and polarity $\pi = +$. From the polarity and degree we furthermore compute a connective *con* via

$$\begin{array}{ll} con = \wedge & \text{if } \delta \text{ is even and } \pi = + \text{ or } \delta \text{ is odd and } \pi = -. \\ con = \vee & \text{if } \delta \text{ is odd and } \pi = + \text{ or } \delta \text{ is even and } \pi = -. \\ con = \otimes & \text{otherwise, i.e., } \pi = \pm \end{array}$$

Then assume that a sub-expression $n : \mathbf{ite}(a, b, c)$ has associated polarity π, degree δ and connective *con*. We assign polarities and degrees to subexpressions a, b, and c using the following rules:

$$\pi(a) := \begin{cases} \pi & \text{if} \quad c \rightsquigarrow b \\ -\pi & \text{if} \quad b \rightsquigarrow c \\ \pm & \text{otherwise} \end{cases}$$
$$\pi(b) := \pi(c) := \pi$$

$$\begin{array}{ll} \text{if } b = \mathbf{true}, con = \wedge \text{ or } b = \mathbf{false}, con = \vee : & \delta(a) := \delta(c) := \delta + 1 \\ \text{if } b = \mathbf{true}, con \neq \wedge \text{ or } b = \mathbf{false}, con \neq \vee : & \delta(a) := \delta(c) := \delta \\ \text{if } c = \mathbf{true}, con = \wedge \text{ or } c = \mathbf{false}, con = \vee : & \delta(a) := \delta(b) := \delta + 1 \\ \text{if } c = \mathbf{true}, con \neq \wedge \text{ or } c = \mathbf{false}, con \neq \vee : & \delta(a) := \delta(b) := \delta \\ \text{if } b, c \notin \{\mathbf{true}, \mathbf{false}\} : & \delta(a) := \delta + 2, \quad \delta(b) := \delta(c) := \delta + 1 \end{array}$$

The use of a digraph representation of expressions implies that the same sub-expression can be visited starting from different paths. In this case the degree is updated to the smallest one and conflicting polarities give rise to a dual polarity marking. Repeated traversal of the same subtree is naturally avoided when a previous degree is not larger and the polarities coincide.

The splitter procedure computes polarity and literal weight simultaneously and returns unit literals whenever possible, otherwise chooses literals of smallest possible weights.

Subexpressions of the form $\forall x : \tau \ . \ n$, $\exists x : \tau \ . \ n$ are skolemized via unit splits.

Heuristics: Well-known heuristics from propositional and constraint satisfaction solving, such as dynamic clause addition and conflict directed backtracking [Pro93] or dilemma rules [SS98] have presently not been added to the implementation.

2.5 Summary

We examined different ways to integrate decision procedures and proposed a constraint-based version of Shostak's approach to allow efficient handling of general constraints. Based on a simple proof-calculus we presented a depth-first refutation search implementation.

Chapter 3

Congruence closure

Equality over a vocabulary of uninterpreted function symbols f, g, ... is axiomatized via

reflexivity			$x = x$
symmetry	$x = y$	$\rightarrow$	$y = x$
transitivity	$x = y \wedge z = x$	$\rightarrow$	$z = y$
congruence	$x = y$	$\rightarrow$	$f(\ldots, x, \ldots) = f(\ldots, y, \ldots)$ for each function f

Satisfiability of a set of ground equalities and disequalities can be decided using congruence closure. Given a subterm-closed set T of terms $\{t_1, \ldots, t_n\}$ and a set $\mathcal{E} \subseteq T \times T$ of equations over T, congruence closure is the process of generating the coarsest partition $\mathcal{C}$ of T satisfying

1. $(s, t) \in \mathcal{E} \;\rightarrow\; s \equiv_{\mathcal{C}} t$.
2. $\overline{s} \equiv_{\mathcal{C}} \overline{t} \;\rightarrow\; f(\overline{s}) \equiv_{\mathcal{C}} f(\overline{t})$.

where

$$s \equiv_{\mathcal{C}} t \;\stackrel{\text{def}}{=}\; \exists C \in \mathcal{C} \;.\; s, t \in C \;.$$

Since $\mathcal{C}$ is a partition it automatically satisfies the equality axioms for reflexivity, symmetry and transitivity. The congruence axioms are satisfied by condition 2. Since $\mathcal{C}$ is the coarsest such partition it is guaranteed to only satisfy the consequences of the equality axioms for the terms in T.

Example: Let $T = \{a,\ b,\ c,\ f(a,b),\ f(b,c),\ g(a),\ f(g(a),b)\}$ and assert $\mathcal{E} = \{a = b,\ b = c\}$. Then the congruence closure $\mathcal{C}$ is the partition:

$$\{\{a,\ b,\ c\},\ \ \{f(a,b),\ f(b,c)\},\ \ \{g(a)\},\ \ \{f(g(a),b)\}\}$$

Thus, the ground constraint:

$$a = b \;\wedge\; b = c \;\wedge\; f(a,b) \neq g(a) \;\wedge\; f(b,c) \neq f(g(a),b)$$

is satisfiable because every pair of terms in disequalities are in different classes, but the constraint

$$a = b \ \wedge \ b = c \ \wedge \ f(a,b) \neq f(b,c) \ \wedge \ f(b,c) \neq f(g(a),b)$$

is unsatisfiable because the terms in the first disequality are equal by congruence closure.

In this chapter we will not only develop new congruence closure algorithms (plenty of efficient versions of these are already around), but also use the congruence closure algorithm to manage the use of other decision procedures.

Shostak's combination of decision procedures uses a congruence closure algorithm to maintain and manage equalities and propagate these through function symbols. A main advantage of this approach is that equality information is kept in one place: in the union-find structure used by the congruence closure algorithm. Everywhere else equalities are propagated using *canonization*. In analogy with term-rewriting, the congruence closure algorithm provides a completion procedure, and equalities are propagated using rewriting into a normal form. It is also easy to process constraints incrementally using the congruence closure based approach, such that inconsistencies can be detected early in a refutation search.

Other congruence closure algorithms are discussed in [CS70, Koz77, NO78, Sho78, McA91]. For the combination of theories various congruence closure algorithms have been proposed in [Sho84, CLS96, BDL96]. They share a common restriction of not being able to handle cyclic terms. This restriction does not apply here, and we will make use of this added feature in Chapter 6. The algorithm does not require a recursive path ordering on solutions in the style of [BDL96], and does not need auxiliary *signature* terms [Sho84] and repeated recursive canonizations. On the less encouraging side, completeness of the congruence closure approach still relies on less than obvious properties of the algorithm, and is highly sensitive to interface compatibility with external solvers.

Summary of Results: In the empty theory where all function symbols are uninterpreted our basic congruence closure algorithm can be tuned to run in average time $\mathcal{O}(n \log(n))$ when processing at most n equalities with a total of n different sub-terms. This is better than other congruence closure algorithms aimed at combining theories, but it is comparable with the best known bound for congruence closure algorithms [DST80]. Our algorithm differs from this by dispensing with a signature table, and uses instead a dynamic array to represent and modify terms. The price consists of using $\mathcal{O}(n \log(n))$ space instead of linear space. The extra space consumption has not been a practical concern for the examples used so far and it has the added benefit of caching intermediary results.

In the term-rewriting community some interest has been devoted in the generation of a ground confluent term-rewriting system from a set of ground equalities. For instance [GNP+93] give an $\mathcal{O}(n^3)$ algorithm for generating such a set. This is improved in [Kap97] to $\mathcal{O}(n^2)$. Naturally, in Section 3.3.2 we notice that our algorithm can be used to generate a ground confluent rewrite system in average time $\mathcal{O}(n \log(n))$.

In Section 3.5 we connect the congruence closure algorithm with the rigid E-unification problem and obtain the nice corollary that a rigid E unifier can be expressed as an ordered

set of pairs in the congruence closure graph.

In Chapter 4 we show how the data structures that are used can also be augmented to propagate relational symbols other than equality.

3.1 Union-find

A partition of equivalence classes can conventiently be maintained and updated using a *union-find* structure. Different union-find algorithms are analyzed in depth in [Tar75].

A union-find structure maintaining a partition over a set of elements $\mathcal{Q}$ uses two functions:

find : $\mathcal{Q} \to \mathcal{Q}$ which maps elements from the same class to a unique representative within that class. When $find(q) = q$, we say that q is a *root node*.

union : $\mathcal{Q} \times \mathcal{Q} \to$ **unit** The function $union(q, r)$ takes two root nodes q and r, and merges their classes by setting $find(r) := q$ and similarly with all other elements in the same class as r.[1]

We extend the union-find structure to also contain a set of edges *Edges* between union-find nodes from Q. Edges are labeled by auxiliary binary predicates, such as $\neq$. In Chapter 4 we treat the case where edges are labeled by binary predicates that represent monotone relations and partial ordering constraints.

For now we augment the precondition of *union* to require that the argument nodes are not connected with an edge labeled by $\neq$. This corresponds to detecting inconsistent disequalities. The effect of $union(q, r)$ now also updates the set of edges *Edges* by repointing edges to and from r to instead enter and leave the new root q. In this way one maintains the invariant that edges only connect root nodes.

To dynamically allocate and keep track of union-find nodes, the union-find structure contains a set Q of allocated nodes, and a function *new* allocating a new node:

Q: $\mathcal{Q}-$**set**	Set of allocated union-find nodes. Initially $Q = \emptyset$.
new : **unit** $\to \mathcal{Q}$	Allocates a fresh state q. The effect is: $Q := Q \cup \{q\}$ for some $q \notin Q$.

3.2 Terms

Terms are maintained on top of the union-find structure by associating each allocated node in Q with a (unique) pair $(f, \langle q_1, \ldots, q_n \rangle)$ where f is a function symbol of arity n (variables and constants are treated as a nullary function symbols), and $q_1, \ldots, q_n \in Q$. More suggestively we write $f(q_1, \ldots, q_n)$ instead of $(f, \langle q_1, \ldots, q_n \rangle)$ to indicate that the arguments of f are the terms associated with $q_1, \ldots, q_n$.

Thus, the domain $\mathcal{T}$ of terms is given by:

[1] The type **unit** is the (trivial) singleton domain.

$\mathcal{T} = \mathcal{F} \times \mathcal{Q}^*$
$\mathcal{F}$ consists of function symbols, constants and skolem variables.

The labeling of $\mathcal{Q}$ is maintained by

$L_{\mathcal{Q}} : Q \to \mathcal{T}$

By keeping $L_{\mathcal{Q}}$ injective there is a function acting as inverse on the range of $L_{\mathcal{Q}}$,

$L_{\mathcal{T}} : \mathcal{T} \to \mathcal{Q}$

It is required to associate each term t in the range of $L_{\mathcal{Q}}$ with the union-find node q such that $L_{\mathcal{Q}}(q) = t$. While the range of $L_{\mathcal{T}}$ is Q (that is, coincides with the domain of $L_{\mathcal{Q}}$), its domain will in general only be required to include $\mathbf{range}(L_{\mathcal{Q}})$. In our implementation $L_{\mathcal{T}}$ is a hash-table mapping terms to array indices, and $L_{\mathcal{Q}}$ is a dynamic array of terms.

$use : Q \to Q-$**set :** We also need to maintain a function that gives a super-set of the union-find nodes that use a given union-find node. That is, we maintain the invariant

$$\{q' \in Q \mid L_{\mathcal{Q}}(q') = f(q_1, \ldots, q, \ldots, q_n)\} \subseteq use(q)$$

When a congruence closure node q is allocated with *new*, the corresponding value of $use(q)$ is initialized as $\emptyset$.

canonical: The congruence closure algorithm works by rewriting terms into canonical form according to the equalities it is supplied with. The boolean tag $canonical(q)$ is used to indicate whether all subterms of the term $L_{\mathcal{Q}}(q)$ are roots with respect to the union-find structure.

$canonical : Q \to \mathcal{B}$

children: As a shorthand we define

$children(q) = \{q_i \mid L_{\mathcal{Q}}(q) = f(q_1, \ldots, q_n) \ \wedge \ i \in \{1, \ldots, n\} \ \}$

3.3 Uninterpreted congruence closure

We will now describe a congruence closure algorithm for uninterpreted function symbols. The core algorithm works by merging union-find nodes and propagating the equality information up through function symbols. It consists of the functions *merge* and *insert* and is given in Figure 3.1.

Informally, *merge* asserts equality of nodes a and b by

```
1.  merge(a, b) =
2.      if a ≠ b then
3.      let
4.          (a, b) = if |use(a)| < |use(b)| then (b, a) else (a, b)
5.      in
6.          union(a, b);
7.          for each u ∈ use(b) where canonical(u) do
8.          let
9.              f(q_1, ..., q_n) = L_Q(u)
10.             t = f(find(q_1), ..., find(q_n))
11.         in
12.             if t ≠ f(q_1, ..., q_n) then
13.                 canonical(u) := false;
14.                 merge(find(u), insert(t, u))

15. insert(t, q) =
16.     if t ∈ dom(L_T) then return find(L_T(t)) else
17.         L_T := L_T † [t ↦ q];
18.         L_Q := L_Q † [q ↦ t];
19.         canonical(q) := true;
20.         for each q' ∈ children(t) do use(q') := use(q') ∪ {q};
21.         return find(q)
```

Figure 3.1: Procedures *merge* and *insert*

1. In line 4 a and b are swapped if the use-list of a is longer than the use-list of b. This causes *union* to make as root the node with the smallest use-list, and gives the average running time claimed in Theorem 3.3.2.

2. To propagate the newly obtained equality information every occurrence of pointers to the non-root b must be replaced by pointers to the root a. Hence each term potentially using b must be updated by the new functionality of *find*.

3. In lines 9 and 10 the terms affected by the new functionality of *find* are generated. Since we allow implementations of the use-list as a list with repetitions, or including terms without occurrences of b, we check in line 12 whether the update caused any changes.

4. *insert*(t, q) first checks if the updated term is already declared and returns the *find* of the node associated with the term in this case. If the updated term t is not already present, then *insert* replaces the previous version of the term labeling q by the new

t. The canonicity of q is reconfirmed, and the use-list of t's children is updated to include q.

5. *merge* is called recursively on the results.

There are two observations worth pointing out concerning *insert*:

1. No new union-find nodes are generated by *merge* and *insert*.
2. The update $L_{\mathcal{Q}} := L_{\mathcal{Q}} \dagger [q \mapsto t]$ in line 17 associates an entirely new term with the node q. The terms however share the same function symbol and only differ in their arguments. By construction, applying *find* to the arguments of the previous term gives the arguments of the new term.

A set of equality constraints

$$\mathcal{E} : s_1 \doteq t_1, \quad s_2 \neq t_2, \quad s_3 \doteq t_3, \quad \ldots, \quad s_n \doteq t_n$$

where s_i, t_i are expressions over $\mathcal{F}$ is processed by first converting the expressions into terms over $\mathcal{T}$ using the *canonize* function, and then invoking *merge* on the resulting equations over $\mathcal{T} \times \mathcal{T}$ while connecting nodes corresponding to disequalities by $\neq$-edges. The function *addConstraints* fails if a node is ever connected to itself with a $\neq$-edge. The utilities for processing equations are shown in Figure 3.2.

$canonize(f(t_1, \ldots, t_n)) = insert(f(canonize(t_1), \ldots, canonize(t_n)), new())$

$addConstraints(\mathcal{E}) =$
for each $(t \doteq s) \in \mathcal{E}$ **do** $merge(canonize(t), canonize(s))$
for each $(t \neq s) \in \mathcal{E}$ **do** connect $canonize(t)$ and $canonize(s)$ by $\neq$

Figure 3.2: Canonization and processing of equalities

Example: Given

$$\mathcal{E} : \{g(f(a)) \doteq w, \quad w \neq g(a), \quad f(a) \doteq a\},$$

a left-to right processing creates the structure (the index on the nodes is not necessarily chronological)

$$\begin{array}{lcll} L_{\mathcal{T}}(a) & = & q_1 & use(q_1) = \{q_2, q_5\} \\ L_{\mathcal{T}}(f(q_1)) & = & q_2 & use(q_2) = \{q_3\} \\ L_{\mathcal{T}}(g(q_2)) & = & q_3 & use(q_3) = \emptyset \\ L_{\mathcal{T}}(w) & = & q_4 & use(q_4) = \emptyset \\ L_{\mathcal{T}}(g(q_1)) & = & q_5 & use(q_5) = \emptyset \end{array}$$

Initially $\mathit{find} := \lambda x.x$, and all nodes are *canonical*. The first equality $g(f(a)) \doteq w$ requires merging q_3 and q_4. This causes the effect

$$\mathit{find} := [q_4 \mapsto q_3]$$

In other words *find* behaves as the identity on q_1, q_2, q_3, and q_5, and maps q_4 to q_3. Asserting the disequality $w \neq g(a)$ causes $\mathit{find}(q_4) = q_3$ and q_5 to be connected by an edge labeled by $\neq$. Thus, the effect is

$$\mathit{Edges} := \{(q_3, \neq, q_5)\}$$

Finally, in asserting $f(a) \doteq a$, q_2 and q_1 are merged causing first the effect

$$\mathit{find} := [q_4 \mapsto q_3,\ q_2 \mapsto q_1]$$

since q_2 has the smallest use-list, and then, since $q_3 \in \mathit{use}(q_2)$ *merge* requires to set:

$$t = g(\mathit{find}(q_2)) = g(q_1) \quad \text{since} \quad g(q_2) = L_{\mathcal{Q}}(q_3)$$

and invoke $\mathit{insert}(t, q_3)$, which since $t \in \mathbf{dom}(L_{\mathcal{T}})$, evaluates to q_5. A recursive invocation $\mathit{merge}(q_3, q_5)$ is now initiated, which requires a call to $\mathit{union}(q_3, q_5)$. This call fails as $(q_3, \neq, q_5) \in \mathit{Edges}$, indicating that the entire set of constraints is unsatisfiable.

3.3.1 Correctness

We will here prove that the congruence closure algorithm is sound and complete. In terms that we will make precise later, this means that invoking *addConstraints* on a set of equalities produces a structure reflecting only the asserted equalities and the consequences of the equality axioms. Equality in the resulting structure is reflected by the functionality of *find*: two terms associated with nodes q and q' are equal if and only if $\mathit{find}(q) = \mathit{find}(q')$.

First note that a set of equalities $\mathcal{E}$ on terms corresponds in a natural way to an initial partition $\mathcal{C}_0$ of a subset of $\mathcal{Q}$. It is obtained by first canonizing every term in $\mathcal{E}$ to get $\mathit{canonize}(\mathcal{E}) \subseteq Q \times Q$, and then making $\mathcal{C}_0$ be the least equivalence class where every pair in $\mathit{canonize}(\mathcal{E})$ is in the same class. We will therefore for convenience work with $\mathit{canonize}(\mathcal{E})$ and the partition $\mathcal{C}_0$ when stating and proving correctness.

In analogy with the definition of congruence closure in the introduction we define

Definition 3.3.1 (Congruence closure on $\mathcal{Q}$) *The congruence closure of any partition $\mathcal{C}_0$ of the set of declared nodes Q is the coarsest partition $\mathcal{C}$ such that*

1. *$\mathcal{C}_0$ is a refinement of $\mathcal{C}$ (i.e., $\forall C \in \mathcal{C}_0 \exists C' \in \mathcal{C}\ .\ C \subseteq C'$.)*
2. *Whenever $t, t' \in \mathbf{dom}(L_{\mathcal{T}})$, $t = f(q_1, \ldots, q_n)$, $t' = f(q'_1, \ldots, q'_n)$, and $q_1 \equiv_{\mathcal{C}} q'_1, \ldots, q_n \equiv_{\mathcal{C}} q'_n$, then $L_{\mathcal{Q}}(t) \equiv_{\mathcal{C}} L_{\mathcal{Q}}(t')$.*

If we process $\mathit{canonize}(\mathcal{E})$ by calling $\mathit{merge}(\mathit{find}(q), \mathit{find}(q'))$ for every pair $(q, q') \in \mathit{canonize}(\mathcal{E})$, then

Theorem 3.3.2 (Termination) *The function merge terminates on all inputs in space and average time*

$$\mathcal{O}(n \log(n)),$$

where n is the number of nodes in the input term-graph (different subterms in the input).

After *addConstraints* has terminated we have:

Theorem 3.3.3 (Soundness)

$$q \equiv_{\mathcal{C}} \mathit{find}(q), \qquad \forall q \in Q \ .$$

Soundness states that *find* respects the congruence relation $\equiv_{\mathcal{C}}$. In particular, if q and r are nodes such that $\mathit{find}(q) = \mathit{find}(r)$, then q and r are in fact congruent: $q \equiv_{\mathcal{C}} r$. Soundness is an obvious property of our algorithm: *merge* only propagates equalities that are implied by $\equiv_{\mathcal{C}}$.

Theorem 3.3.4 (Completeness)

$$\mathit{If} \quad q \equiv_{\mathcal{C}} r \quad \mathit{then} \quad \mathit{find}(q) = \mathit{find}(r) \ .$$

Completeness means that *find* collapses all congruence classes in $\mathcal{C}$. To establish completeness requires a more careful analysis. For this purpose consider the version of *merge* in Figure 3.3 augmented with auxiliary variables U and V, which are initially the empty sets.

Informally, U consists of the set of nodes where *find* does not act as identity any longer as a consequence of the call to *union* in line 6. The set V consists of the nodes whose terms contain an element from U. To accommodate for the delay in updating *find* of the *use* set of b we therefore define

$$\mathit{find}_U(q) \stackrel{\text{def}}{=} \textbf{if } q \in U \textbf{ then } q \textbf{ else } \mathit{find}(q) \ .$$

Completeness now relies on the following invariants whose conjunction is inductive.

Invariant 3.3.5 *If $f(q_1, \ldots, q_n) \in \mathbf{dom}(L_{\mathcal{T}})$ then*

$$f(\mathit{find}_U(q_1), \ldots, \mathit{find}_U(q_n)) \in \mathbf{dom}(L_{\mathcal{T}}).$$

Invariant 3.3.6 *If $L_{\mathcal{Q}}(q) = f(q_1, \ldots, q_n)$ and $q \in V$ then*

$$\mathit{canonical}(q) \quad \mathit{if\ and\ only\ if} \bigwedge_{i=1}^{n} q_i = \mathit{find}_U(q_i).$$

Invariant 3.3.7 *If $L_{\mathcal{Q}}(q) = f(q_1, \ldots, q_n)$ and $q \notin V$ then*

$$\mathit{canonical}(q) \quad \mathit{if\ and\ only\ if} \bigwedge_{i=1}^{n} q_i = \mathit{find}(q_i).$$

```
1.  merge(a,b) =
2.      if a ≠ b then
3.      let
4.          (a, b) = if |use(a)| < |use(b)| then (b, a) else (a, b)
5.      in
6.          union(a, b);
7.          [U := U ∪ {b}; V := V ∪ {u ∈ use(b) | canonical(u)};]
8.          for each u ∈ use(b) where canonical(u) do
9.          let
10.             f(q1, ..., qn) = L_Q(u)
11.             t = f(find(q1), ..., find(qn))
12.         in
13.             if t ≠ f(q1, ..., qn) then
14.                 canonical(u) := false; [V := V \ {u};]
15.                 merge(find(u), insert(t, u))
16.         [U := U \ {b}]
```

Figure 3.3: Augmented version of *merge*

Invariant 3.3.8 *If* $f(q_1, \ldots, q_n) \in \mathbf{dom}(L_{\mathcal{T}})$ *then*

$$find(L_{\mathcal{T}}(f(q_1, \ldots, q_n))) = find(L_{\mathcal{T}}(f(find_U(q_1), \ldots, find_U(q_n)))).$$

When the congruence closure algorithm has terminated $U = \emptyset$ and $V = \emptyset$, thus, $find_U = find$.

Proof of 3.3.4:

A simple way to construct the equivalence relation $\equiv_{\mathcal{C}}$ is by computing the least fix-point obtained by starting with $\equiv_{\mathcal{C}_0}$, which is the equivalence class obtained from the input equations. Inductively, the $i+1$'st partition $\equiv_{\mathcal{C}_{i+1}}$ is obtained from $\equiv_{\mathcal{C}_i}$ by taking the coarsest partition satisfying

1. If $s \equiv_{\mathcal{C}_i} t$ then $s \equiv_{\mathcal{C}_{i+1}} t$.
2. If $\overline{s} \equiv_{\mathcal{C}_i} \overline{t}$ then $f(\overline{s}) \equiv_{\mathcal{C}_{i+1}} f(\overline{t})$.

The final partition $\equiv_{\mathcal{C}}$ is then $\equiv_{\mathcal{C}_n}$ for some (finite) n because there are only finitely many terms.

We shall establish by induction on $i \leq n$, that whenever $q \equiv_{\mathcal{C}_i} r$ then $find(q) = find(r)$.

1. If $q \equiv_{\mathcal{C}_0} r$, then q and r are connected via equalities in $\mathit{canonize}(\mathcal{E})$ that are merged in some order. The merging causes eventually that *find* is updated such that $\mathit{find}(q) = \mathit{find}(r)$.
2. Deriving $q \equiv_{\mathcal{C}} r$ using symmetry, reflexivity and transitivity is direct since a partition induced by *find* is automatically an equivalence class.
3. The only other way we can derive $q \equiv_{\mathcal{C}_{i+1}} r$ is by the existence of $t, s \in \mathbf{dom}(L_{\mathcal{T}})$, $t = f(q_1, \ldots, q_n)$, $s = f(r_1, \ldots, r_n)$, and $q_1 \equiv_{\mathcal{C}_i} r_1, \ldots, q_n \equiv_{\mathcal{C}_i} r_n$, such that $q = L_{\mathcal{T}}(t)$, $r = L_{\mathcal{T}}(s)$. The induction hypothesis asserts that $\mathit{find}(q_1) = \mathit{find}(r_1)$, $\ldots$, $\mathit{find}(q_n) = \mathit{find}(r_n)$. From the invariants 3.3.5 and 3.3.8 used for both t and s we obtain that $\mathit{find}(L_{\mathcal{T}}(q)) = \mathit{find}(L_{\mathcal{T}}(r))$.

▟

Proof of 3.3.2:

We use same idea that is in [DST80] by "processing the smaller half". This requires *union* to choose as root the state with the largest use-list, such that the for-loop processes the smaller half only. The analysis follows the proof in [DST80] closely: (1) every time *merge* is called, one equivalence class is eliminated, (2) accessing the union-find structure $\mathcal{O}(n)$ times takes time $\mathcal{O}(n \log n)$ since the choice of the union root is dictated by the length of the use-list (which is stored as a doubly linked list with possible repetitions and an integer length), but the find-structure can be dynamicaly updated during the find operations.

▟

A closer comparison with the Downey, Sethi, Tarjan algorithm is in order. Under the assumption that $L_{\mathcal{T}}$ is implemented using a hash-table, [DST80] require on average $\mathcal{O}(n \log n)$ time and $\mathcal{O}(n)$ space. The present algorithm uses more space. The first difference is that the other algorithm recomputes a *signature table* in each iteration and deletes entries after use. Here, all terms are kept in the hash-table. The other difference is that the other algorithm lists the use list for all nodes in the same equivalence class, whereas here only the use-list associated with the root is listed. Instead, the present algorithm ensures that new terms are generated for children of the root and inserted using *insert* into the use list of the root.

3.3.2 Ground rewriting

It is now simple to extract a confluent rewrite system from the $L_{\mathcal{T}}$ and $L_{\mathcal{Q}}$ after a set of equalities have been processed. The rewrite system has the same effect as *canonize* and will therefore be able to detect inconsistent disequalities. We extract the rewrite system as follows:

1. For each $q \in \mathbf{dom}(L_{\mathcal{Q}})$ introduce a fresh constant symbol C_q.

2. For each $q \in \mathbf{dom}(L_{\mathcal{Q}})$ where $\mathit{canonical}(q)$ and $f(q_1, \ldots, q_n) = L_{\mathcal{Q}}(q)$, add the rewrite rule:

$$f(C_{q_1}, \ldots, C_{q_n}) \longrightarrow C_{\mathit{find}(q)} \ .$$

When $\mathit{canonical}(q)$ holds each immediate sub-term is labeled by roots so the rewrite system does not have any critical pairs and is obviously terminating as the rewrite rules only replace old functions with the fresh constants. The fresh constants, on the other hand, are never at the top-level of the left hand side of the rewrite rules.

3.4 Congruence closure with theories

For incorporating theories in the congruence closure algorithm we shall not diverge much in spirit from Shostak's approach. The resulting algorithm is on the other hand significantly more compact than Shostak's, but perhaps still subtle. Figure 3.4 presents the modified version of *merge* (from Figure 3.1). The modifications use new auxiliary functions *solve* and σ described below. One important change is that *merge* is no longer allowed to swap its arguments because the directed union now has to be consistent with the output of *solve*.

```
1.  merge(a, b) =
2.      if a ≠ b then
3.          union(a, b);
4.          for each u ∈ use(b) where canonical(u) do
5.          let
6.              f(q_1, ..., q_n) = L_Q(u)
7.              t = σ(f(find(q_1), ..., find(q_n)))
8.          in
9.              if t ≠ L_Q(u) then
10.                 canonical(u) := false;
11.                 if not interpreted(u)
12.                     then process(solve(find(u), insert(t, u)))
13.                 else if u = find(u)
14.                     then merge(u, insert(t, u))

15. process(θ) =
16.     for each [q_1 ↦ q_2] ∈ θ do merge(find(q_1), find(q_2))
```

Figure 3.4: Merge in the presence of theories

The canonizer $\sigma : \mathcal{T} \to \mathcal{T}$: The modified *merge* uses a *canonizer* σ to normalize terms with respect to the updates of *find* and rewrite the resulting term into normal form.

High-level requirements for the canonizer are described in Section 2.2.3. For convenience we are deviating from the explicit signature for σ from Section 2.2.4 and leave the context argument and update implicit.

While the canonizer may need to access $L_{\mathcal{Q}}$ to normalize sub-terms of the normalized $f(\mathit{find}(q_1), \ldots, \mathit{find}(q_n))$ a full recursive canonization of this argument is not necessary, as *merge* provides eventual canonization of these sub-terms and upwards propagation via the *use*-lists. In our implementation, however, the theory-specific solvers do use *find* on sub-terms of interpreted arguments to minimize repeated work.

A note of subtlety: In line 17 of the *insert* function, when t is a term that does not already exist in $L_{\mathcal{T}}$, the operation reuses the node u to represent t. The following call to *merge* then has no effect. For almost all reasonable theories this property causes incompleteness, as terms are canonized differently according to the form of their sub-terms.

For instance, if y is merged with x, and y occurs in $y+z$ and $y+z$ occurs in $-x+(y+z)$, then first the term $y+z$ is replaced by the fresh term $x+z$. Since $x+z$ is fresh, the same union-find node is used for it and the change is not propagated to the super-term $-x+(x+z)$, which canonizes differently.

To avoid this incompleteness we require that the solver allocates fresh union-find nodes whenever it returns an interpreted term (alternatively we can use an alternative *insert*, at the expense of adding additional pseudo code).

solve : $\mathcal{Q} \times \mathcal{Q} \to (\mathcal{Q} \times \mathcal{Q})^*$**:** In invoking $\mathit{solve}(q, q')$ the solver calls *addConstraints* with the current context of constraints and equality constraint $q = q'$. It returns the set of derived equalities in form of a substitution θ and an updated context of constraints. If the updated context of constraints is trivially unsatisfiable we interrupt the iterated calls to *merge* and return *false*.

We do not require that the substitution be idempotent. This is essential to handle cyclic data-structures. The substitutions should rather correspond to a most general unifier.

interpreted : $\mathcal{Q} \to \mathcal{B}$**:** Each theory determines which terms are interpreted. In the theory of linear arithmetic, terms whose main function symbol is $+$ or $-$ are interpreted. Diverging from other approaches we shall not treat data-type selectors (and record projectors) as interpreted function symbols to obtain heuristic speed-up and to be able to handle satisfiable cyclic constraints, such as $\mathtt{CAR}(\mathtt{NIL}) = \mathtt{NIL}$.

Comparisons with [Sho84]: Shostak's congruence closure uses an auxiliary function *canonsig* to recursively canonize sub-terms. This function is absent from our algorithm because the use of a shared term structure enables canonization to eventually be propagated up through terms. We note the following differences:

- Shostak's algorithm diverges when the interpreted theory allows cyclic data-structures because *canonsig* calls itself recursively on arguments of interpreted function symbols.
- *canonsig* is called twice, both before invoking *solve*, and after invoking *solve*. We have not found this to be necessary.

- In Shostak's data-type solver, selectors are treated as interpreted function symbols, and may therefore not be in the domain of the result of *solve*. This causes unsound and expensive creation of new variables in solved forms.

Comparisons with [BDL96]: The elegant algorithm from [BDL96] requires solutions $x \mapsto t$ to satisfy that x is not a subterm of t. However, this requirement prevents a partial elimination of variables in non-linear constraints. For instance, given the equality:

$$y \quad = \quad x \cdot f(y) \tag{3.1}$$

our approach can detect an inconsistency among

$$f(y) > 0 \ \wedge \ y \cdot x < 0 \ . \tag{3.2}$$

If we replace y by $x \cdot f(y)$ in the constraints we obtain the obviously unsatisfiable

$$f(y) > 0 \ \wedge \ x^2 \cdot f(y) < 0$$

without having to maintain the eliminated equality. On the other hand, if we were to require that all solutions satisfy the subterm relationship, then the equality (3.1) cannot be eliminated in establishing inconsistency of (3.2).

Naturally, the same restriction to non-cyclic data structures also applies to SVC, though these may not necessarily be interesting for the domain of SVC.

3.5 Rigid E-unification

In this Section, we first present a definition of rigid E-unification. By reformulating a decidability proof for rigid E-unification using the data-structures from the previous sections, we arrive at Corollary 3.5.4, which states that a rigid E-unifier can be found by guessing a set of pairs from the union-find nodes Q. This gives a neat reformulation of rigid E-unification as a simple constraint satisfaction problem.

Definition 3.5.1 (Rigid E-unification) *Let $\overline{x}$ be a set of Skolem variables, and let φ be a Horn formula of the form*

$$\underbrace{s_1 = t_1 \wedge \ldots \wedge s_n = t_n}_{E} \ \rightarrow \ s = t \tag{3.3}$$

whose free variables are in $\overline{x}$. The substitution θ with domain $\overline{x}$ is a rigid E-unifier *for φ if θ applied to φ, i.e.,*

$$(s_1 = t_1 \wedge \ldots \wedge s_n = t_n \ \rightarrow \ s = t)\theta$$

is ground valid (i.e., the equality $s = t$ follows from congruence closure with respect to the assumed equalities).

Example: The substitution $[x \mapsto c, y \mapsto a]$ is a rigid E-unifier for

$$f(x) = g(a) \ \wedge \ h(b) = f(c) \quad \rightarrow \quad g(y) = h(b) \ .$$

The substitution $[x \mapsto b, y \mapsto g(a)]$ is a rigid E-unifier for

$$h(a) = a \ \wedge \ h(x) = a \ \wedge \ h(b) = f(y) \ \rightarrow \ y = g(f(y)) \ .$$

The substitutions $[x \mapsto g^n(a)]$ are all rigid E-unifiers for

$$g(a) = a \ \rightarrow \ x = a \ .$$

Theorem 3.5.2 *[GNRS92, dK94] The rigid E-unification problem is NP-complete.*

In fact de Kogel [dK94] never claims that his reconstruction of the decidability result for rigid E-unification also establishes NP-completeness, but it does, even contrary to the claims in [Bec98]. In his proof a rigid E-unifier is classified as either *connecting*, *non-connecting and reducible*, or *non-connecting and irreducible*. Unifiers (substitutions) are represented in triangular form, such that the triangular substitution $t\langle x_1 \mapsto t_1, x_2 \mapsto t_2, \ldots, x_n \mapsto t_n\rangle$ is applied in stages: $(\ldots((t[x_1 \mapsto t_1])[x_2 \mapsto t_2])\ldots[x_n \mapsto t_n])$. We shall define *reducible* and *connecting* relative to the utilities introduced for our congruence closure algorithm.

Definition 3.5.3 *Let E be a set of equalities and Q be the union-find nodes in the congruence closure after E has been processed.*

- *The map $\langle x_i \mapsto v_i \mid i = 1, \ldots, n \rangle$ is* connecting *if canonize(x_i), and canonize(v_i) are already in Q for each i.*
- *The map $\langle \ldots, x \mapsto u, y \mapsto v, \ldots \rangle$ is* reducible *if canonize(u) is in the transitive closure of use and find from canonize(v).*

To prove decidability of rigid E-unification de Kogel notes that a rigid E-unifier θ for (3.3) can be classified by one of the following conditions:

1. θ is non-connecting and reducible. Then a rigid E-unifier can be found with smaller terms. By repeatedly eliminating reducible pairs, only cases 2 and 3 need to be considered.
2. θ is non-connecting and irreducible. Then by deleting a non-connecting pair $x \mapsto t$ (there is at least one), from the substitution we obtain a triangular form of smaller size.
3. Finally we arrive at a connecting rigid E-unifier. In this case θ can by definition 3.5.3 be equivalently expressed as an ordered list of pairs of congruence closure nodes.

A detailed proof justifying these observations is given in [dK94] and will not be repeated here. The main observation we make in connection with the formulation based on the congruence closure structure is, however:

Corollary 3.5.4 (From congruence closure to Rigid E-unification) *Let Q be a set of union-find nodes after processing all equalities in E and having canonized the terms s and t to be unified. A rigid E-unifier is an ordered set of pairs of Q corresponding to a triangular substitution. The set of pairs can be guessed and checked in polynomial time.*

Guessing such a triangular substitution can be done in $\mathcal{O}(|Q|)$ by listing an ordered set of pairs. The ordered set of pairs then has to be checked for corresponding to a well-formed triangular substitution by unfolding nodes to variables in the domain and terms in the range, pair by pair. Finally, it can be checked for being a rigid E-unifier using the congruence closure algorithm by simply merging the nodes in the substitution.

3.6 A benchmark example

Refinement proofs for pipelined CPUs [BD94] are rather stressful benchmarks for reasoning about uninterpreted functions, McCarthy's update axiom

$$read(write(A, i, e), j) \quad = \quad \textbf{if} \;\; i = j \;\textbf{then}\; e \;\textbf{else}\; read(A, j),$$

and boolean combinations of equalities. To test the congruence closure without using the repository of well-established guiding techniques (see for instance [LO97]) the formulation from [HSG98] was taken without the auxiliary rewrite rules specific for the PVS verifier. The validity checker was then allowed to spend its time on it. For the main correctness claim, nearly one million case splits were required, using a particular splitting heuristic and the entire verification took 150 minutes. Thus, on average, 100 branches were covered each second. Each case required the processing of a structure with 400 different sub-terms, though the incremental accumulation of constraints means that the entire structure did not have to be recreated for each branch.

3.7 Summary

We presented a simple and fast congruence closure algorithm that was extended to integrate decision procedures. We also demonstrated how the congruence closure corresponded to ground completion. The connections with rigid E-unification were highlighted.

Chapter 4

Special relations

We use the term *special relation* for certain monotonicity properties binary relational symbols that enjoy with respect to selected functions. In the basic form we can record a special relationship between predicates $\prec_1$ and $\prec_2$ relative to the function symbol f whenever the monotonicity axiom

$$x \prec_1 y \;\rightarrow\; f(\ldots, x, \ldots) \prec_2 f(\ldots, y, \ldots) \tag{4.1}$$

holds. Anti-monotonicity (where x and y are exchanged in the conclusion) axioms are also considered sufficiently special. Special relations have been studied within resolution theorem proving in for instance [MW86, MSW91, MW92], and [BG95].

The notion of special relations used here is based on theories axiomatizable using Horn clauses. A Horn clause is a disjunction of literals, containing at most one positive literal. Equality itself can be viewed as a special relation as we saw in Chapter 3. The restriction to Horn clauses ensures the existence of an initial model satisfying the implications. The queries that we wish to resolve against a theory axiomatized using Horn clauses are in the simplest case ground formulas. In Chapter 3 we examined a very special relation, equality, and gave an optimized decision procedure for it.

Even simple instances of this scheme cannot be algorithmically decided: Horn clauses with binary relations can be used to encode reachability problems for Turing machines. In more restricted cases we can not only decide ground satisfiability, but also solve the *rigid $\mathcal{T}$-unification* problem, namely whether there is a substitution θ from free variables in φ, such that $\mathcal{T} \models \varphi\theta$. The existence of a rigid $\mathcal{T}$-unifier for φ implies that $\mathcal{T} \models \exists * \varphi$, while the converse is not necessarily the case. Rigid $\mathcal{T}$-unifiers can be used to close branches in a tableau search by providing instantiations of existential force quantifiers.

Results: We propose ground decision support and rigid $\mathcal{T}$-unification problems for two prominent special relations: Partial orders, and monotone relations given by axioms of the form (4.1). The ground decision support is provided as a tight extension of congruence closure and we reuse the data structures developed in Chapter 3.

The results developed here apply to theories that are empty apart from the special relation axioms. In Chapter 6 we take a look at integrating monotone relations in richer

theories.

4.1 Partial orders

A structure is partially ordered by the binary predicate $\preceq$ if it satisfies the three axioms for every x, y, and z

$$\begin{aligned} x \preceq y \ \wedge\ y \preceq z \ &\rightarrow\ x \preceq z && \text{(PO1)} \\ x \preceq y \ \wedge\ y \preceq x \ &\rightarrow\ x = y && \text{(PO2)} \\ &\qquad\ x \preceq x && \text{(PO3)} \end{aligned}$$

A set L_0 of literals can be checked for satisfiablity with respect to the axioms for partial orders and equality together by extending L_0 to a *maximally consistent* set L closed with respect to the axioms for equalities and partial orders. The set of literals L is *maximally consistent* if for all sub-terms $s, t \in L$ either $s \preceq t$ or $s \not\preceq t$, and L is saturated with respect to the equality and partial ordering axioms. Since there are only finitely many subterms in any set of literals L_0 it follows easily that the satisfiability problem for the set of ground literals L_0 is decidable. Using this approach we arrive rather painlessly at a decision procedure for ground formulas with equality and partial orderings.

On the other hand, the full first-order theory of partial orderings reduces to the theory of a binary, symmetric and irreflexive predicate $P(x, y)$ and is therefore undecidable [ELTT65][1]. Indeed, to model $P(x, y)$ using a partial ordering $\prec$ in the closed formula $\varphi(P)$ replace P everywhere by

$$\lambda(x, y)\ .\ \exists z, u\ .\ bot(z)\ \wedge\ z \prec u\ \wedge\ u \prec x\ \wedge\ u \prec y\ \wedge\ x \neq y$$

where $bot(z) = (\forall v\ .\ v \not\prec z)$, $top(z) = (\forall v\ .\ z \not\prec v)$, relativize all quantifiers $\exists x\ .\ \psi$ in φ to $\exists x\ .\ top(x)\ \wedge \psi$, and finally produce $\psi\ \rightarrow\ \varphi$, where ψ restricts all elements to be either without successors, without predecessors, or connecting precisely two elements according to P (the rôle of the auxiliary u in the replacement of P above):

$$\begin{aligned} & \forall x, y, z, u\ .\ x \prec y\ \wedge\ x \prec z\ \wedge\ x \prec u\ \rightarrow\ y = z\ \vee\ y = u\ \vee\ u = z \\ \psi: \quad \wedge\ & \forall x\ .\ \neg bot(x)\ \wedge\ \neg top(x)\ \rightarrow\ (\exists y, z\ .\ y \neq z\ \wedge\ x \prec y\ \wedge x \prec z) \\ \wedge\ & \forall x, y\ .\ x \prec y\ \rightarrow bot(x)\ \vee\ top(y) \end{aligned}$$

We have now obtained a predicate in the theory of partial orderings which is valid if and only if the corresponding predicate over a binary, symmetric, irreflexive relation (aka. the theory of undirected simple graphs) is valid.

On the other hand, the $\forall\exists$-fragment (Π_2-fragment) of the theory of elementary relations is co-NP complete as established in [CC90]. In more detail, the $\forall\exists$-fragment consists of

[1] However, note that the theories of linear and dense linear orderings (with and without end-points) is decidable [CK90, Hod93].

closed formulas of the form

$$\forall\overline{x}\exists\overline{y}\varphi(\overline{x},\overline{y})$$

where φ is a quantifier-free formula whose atomic formulas are of the form

$$R^{(n)} = S^{(n)}, \quad x = y, \quad R^{(n)}(x_1,\ldots,x_n)$$

where R and S are n-ary relation symbols. Since the axioms PO1-PO3 can be added as assumptions to a Π_2-formula without changing the quantifier prefix we obtain that the Π_2-fragment of the theory of partial orders is co-NP complete. While this result in principle gives a singly exponential time algorithm for deciding Π_2 formulas we do not have any goal-directed procedure at hand (which is important in heuristically narrowing the search space). The restriction to pure relational symbols (and not admitting function symbols) also limits the direct applicability of the Π_2-theory of elementary relations.

4.1.1 A ground decision procedure for partial orders

The saturation based approach to check a set of literals L_0 for ground satisfiability in the theory of partial orders suffers from two problems: (1) the saturation includes all combinations of partial ordering constraints on the available terms thereby requiring a quadratic number of predicates in the number of terms, (2) it only gives a non-determinstic procedure for saturating a satisfiable L_0 to a saturated L. If we furthermore extend the vocabulary with the derived relation $x \prec y := x \preceq y \ \wedge \ x \neq y$, we no longer have a Horn axiomatization. Naive tableau rules on the extended language then include splitting (β) rules of the form

$$\frac{x \preceq y}{x = y \mid x \prec y}$$

to obtain a complete decomposition of all atoms.

We address problem (1) by formulating a system where only required constraints are derived. Furthermore constraints implicitly present by the transitivity of partial orderings are not represented explicitly. This only gives a heuristic space saving as for instance the partial order of n elements where element i is connected to $i+1,\ldots,n$, for $i = 1,\ldots,n$ requires the worst case $\binom{n}{2}$ explictly maintained relations, however, as measured in the size of the input the approach does not require any more space. Problem (2) is addressed by maintaining the predicates $=$, $\neq$, $\preceq$ and $\not\preceq$ only. Equality is handled by congruence closure, $\neq$ labels the undirected edges introduced in Section 3.1. This set of edges is updated to contain also the possible labelings $\preceq$ and $\not\preceq$, resulting in a *transitivity* graph.

New edges are added incrementally to *Edges* in the union-find structure as follows:

$q_1 \preceq q_2$ **($q_1 \neq q_2$, $q_1 \not\preceq q_2$):** Add an edge to *Edges* between q_1 and q_2 labeled $\preceq$ (resp. $\neq$, $\not\preceq$).

$q_1 \prec q_2$**:** Add edges for both $q_1 \preceq q_2$ and $q_1 \neq q_2$.

$q_1 \not\preceq q_2$**:** Add edges for both $q_1 \not\preceq q_2$ and $q_1 \neq q_2$.

Total orders are a special case, where constraints of the form $(t_1 \not\preceq t_2)$ are treated as $t_2 \preceq t_1$, so $\not\preceq$ edge labels are not used.

Whenever adding an edge from q to q' labeled by $\not\preceq$, we search for a path of $\preceq$ edges from q to q' labeled by $\preceq$ edges. If such a path exists, all vertices on the path are merged. On the other hand, when a new $q \preceq q'$ edge is added a standard depth-first traversal can be used to search for a path from q' to q consisting of $\preceq$ edges. In the presence of such a path q and q' are merged. To keep the graph minimal in adding a $q \preceq q'$ edge we also need to merge q and q' if there is a $\preceq$-path from q' to q_1 and from from q_2 to q, where $q_2 \not\preceq q_1$ is an edge.

A decision procedure results from the following observation: a conjunction of inequality constraints is unsatisfiable in the theory of partial orders iff its associated transitivity graph is collapsed into a *contradictory graph*, one that contains a $(v, \neq, v)$ edge.

Eager Equational completeness: On the other hand, one can directly extract a model from a non-contradictory graph where all distinct vertices corresponds to different elements. Thus, the decision procedure given here is eagerly complete (see Section 2.2.4).

4.1.2 The rigid *PO*-unification problem

We will here extend the results presented in Section 3.5 on rigid E-unification to rigid unification problems with partial orders.

Definition 4.1.1 (Rigid *PO*-unification) *Let $\overline{x}$ be a set of variables, and let φ be a horn-formula of the form*

$$\bigwedge_{i \in I} t_i \preceq s_i \ \wedge \ \bigwedge_{j \in J} u_j \not\preceq v_j \quad \rightarrow \quad s = t \tag{4.2}$$

whose free variables are in $\overline{x}$. The substitution θ with domain $\overline{x}$ is a rigid PO unifier *for φ if $\varphi\theta$ is ground valid.*

Equivalently we can phrase the rigid PO-unification problem in terms of finding a substitution establishing unsatisfiability of a conjunction

$$\psi : \quad \bigwedge_{i \in I} s_i \preceq t_i \ \wedge \ \bigwedge_{j \in J} u_j \not\preceq v_j \ \wedge \ \bigwedge_{k \in K} w_k \neq z_k \ . \tag{4.3}$$

Equalities, $\prec$ and $\not\preceq$ relations have been eliminated using rewrites from $s = t$ to $s \preceq t \wedge t \preceq s$, together with those from Section 4.1.1. The alternative formulation uses more disequalities, but only one disequality is required to exhibit unsatisfiability as the theory of partial orders is stably infinite (see definition 2.2.1). In other words, the conjunction 4.3 has a rigid PO-unifier if and only if there is a $k \in K$, such that ψ', where K has been set to $\{k\}$, has a rigid PO-unifier.

We also obtain the following result as an extension of NP-completeness for rigid E-unification:

Theorem 4.1.2 *The rigid PO-unification problem is NP-complete.*

Proof outline:

The proof requires essentially no new ideas besides those that can be found in [dK94]. The only crucial dependency on the properties of congruence closure is in eliminating non-connecting irreducible unifiers. ∎

4.1.3 A heuristic for obtaining PO-refuting substitutions

The transitivity graph is not only able to detect ground unsatisfiability, but can also serve as a guide for finding PO-refuters. We say there is a $\prec$-*edge* from u to v if $(u, \preceq, v)$ and $(u, \neq, v)$ are edges in $\mathcal{G}$. To find PO-refuting substitutions for a set of equalities and inequalities, one can find pairs of vertices $\langle v, w\rangle$ that are connected by a $\prec$-edge in the transitivity graph. If E is the set of known equalities at this point, a substitution θ such that $v\theta = w\theta$ under the equalities $E\theta$ is a PO-refuter; that is, θ should be a *rigid E-unifier* [GNRS92] of v and w.

This approach to finding PO-refuters is clearly not complete. A more thorough but still incomplete approach is to consider a pair $\langle v_1, w_1\rangle$ connected by a $\preceq$-path containing a $\prec$-edge, and another pair $\langle w_2, v_2\rangle$ connected by a $\preceq$-path. A substitution θ that is a rigid E-unifier for $\{v_1 = v_2, w_1 = w_2\}$ will also be a PO-refuter.[2]

In the general case, the transitivity graph can be searched to find a sequence of paths and a unifier θ that concatenates the paths into a loop containing a $\prec$-edge. Since θ will be obtained incrementally in a congruence closure context, we define the following:

Definition 4.1.3 *Given a substitution θ and a congruence closure structure CC, a substitution φ is a* θ-compatible rigid CC-unifier *of congruence classes v_1 and v_2 iff φ is less general than θ and φ is a rigid E-unifier of t_1 and t_2 for some $t_1 \in v_1$ and $t_2 \in v_2$, where E is the set of equations implicit in CC. We write $E\_mgus(CC, \theta, v_1, v_2)$ for a set of θ-compatible rigid CC-unifiers of v_1 and v_2.*

Rigid E-unification is NP-complete [GNRS92]. In practice, we are content with quickly identifying a subset of the rigid E-unifiers.We collect E-unifiers using a fast, but again incomplete, test to eliminate redundant substitutions.

The procedure *EXPAND* defined below updates a set S of $PO-$refuting substitutions for the theory of partial orders. It searches the set of paths in the transitivity graph examining one sequence of paths at most once. $TC(v)$ denotes the $\preceq$-*transitive closure* from vertex v, i.e., the set of vertices reachable from v by $\preceq$-edges. $TC^+(v)$ is $TC(v) \setminus \{v\}$. $\mathcal{V}(\mathcal{G})$ is the set of vertices of $\mathcal{G}$. $\mathcal{CC}(\mathcal{G})$ is the congruence closure structure associated with $\mathcal{G}$. After the invocation *EXPAND* $(\mathcal{G}_0, v_0, v_0, [\,])$, each recursive call *EXPAND* $(\mathcal{G}, v_1, v_2, \theta)$ maintains the invariants (a) $v_2 \in TC(v_1)$ in $\mathcal{G}$, and (b) $\mathcal{G}$ is obtained from $\mathcal{G}_0$ by asserting the equalities given by θ. This is ensured by the function *add_substitution*, which merges nodes and

[2]Edges labeled $\not\prec$ and single $\neq$ edges can be similarly used to obtain PO-refuting substitutions. To simplify the exposition, we omit these cases.

collapses the graph as described above. The vertices v_1' and v_4' are the counterparts of v_1 and v_4 in $\mathcal{G}'$. *EXPAND* must terminate, since the size of V_1 decreases with each recursive call.

$S \leftarrow \emptyset$; $EXPAND\,(\mathcal{G}_0, v, v, [\,])$; **return** S; where:
$EXPAND\,(\mathcal{G}, v_1, v_2, \theta) =$
  $V_1 \leftarrow \mathcal{V}(\mathcal{G}) \setminus TC^+(v_1)$
  $V_2 \leftarrow TC^+(v_2)$
  for each $(v_3, v_4) \in V_1 \times V_2$ **do**
    $S' \leftarrow E\_mgus(\mathcal{CC}(\mathcal{G}), \theta, v_3, v_4)$
    for each $\theta' \in S'$ **do**
      $\mathcal{G}' \leftarrow add\_substitution(\theta', \mathcal{G})$
      if $\mathcal{G}'$ is a contradictory graph
        then $S \leftarrow S \cup \{\theta'\}$
        else $EXPAND\,(\mathcal{G}', v_1', v_4', \theta')$

In the worst case, *EXPAND* will search exponentially many paths. However, the moderate size of transitivity graphs arising from typical verification conditions, and the incremental unification restriction, make the procedure practical.

An example: The validity of

$$(\forall x.(x \preceq y \rightarrow P(x))) \wedge (\forall u.\exists z.z \prec u) \rightarrow \exists v.(v \prec y \wedge P(v))$$

is established in 0.07 seconds using the rigid PO-unifier $[v \mapsto f_z(u),\ x \mapsto f_z(u)]$ which can be found using the search procedure.

4.2 Transitive relations

A simpler case than partial orders is that of transitive relations. A relation R is transitive if it satisfies the transitivity axiom

$$R(x,y) \ \wedge \ R(y,z) \quad \rightarrow \quad R(x,z) \qquad \text{(T)}$$

4.2.1 Rigid T-unification

When R is a transitive relation, we define the *rigid T-unification* problem as follows:

Definition 4.2.1 (Rigid T-unification) *Let $\overline{x}$ be a set of variables, and let φ be a horn-formula of the form*

$$s_1 = t_1 \wedge \ldots \wedge s_n = t_n \wedge \bigwedge_{i=1}^{m} R(u_i, v_i) \ \rightarrow \ R(s,t) \tag{4.4}$$

whose free variables are in $\overline{x}$. The substitution θ with domain $\overline{x}$ is a rigid T-unifier for φ if $\varphi\theta$ is ground valid.

Proposition 4.2.2 *The rigid T-unification problem is NP-complete.*

Proof:

NP-hardness: Take an instance of rigid E-unification

$$\bigwedge_{i=1}^{m} s_i = t_i \;\rightarrow\; s = t \tag{4.5}$$

and translate it to:

$$s_1 = t_1 \wedge \ldots \wedge s_n = t_n \wedge R(s,t) \;\rightarrow\; R(t,s) \tag{4.6}$$

Membership in NP: To get $R(s,t)$ in the consequent of formula (4.4) the premises of the implication must provide an R-path. The substitution must provide the merging of states along this path, and thus establish a number of equalities $s = u_{i_1}$, $v_{i_j} = u_{i_{j+1}}$, $v_{i_k} = t$. Thus, it suffices to guess a path of length at most m (there are m conjuncts of R in the premise) and verify the following rigid E-unification instance, where h is a fresh function symbol:

$$s_1 = t_1 \wedge \ldots \wedge s_n = t_n \rightarrow h(s, v_{i_1}, \ldots, v_{i_k}) = h(u_{i_1}, \ldots, u_{i_k}, t) \;. \tag{4.7}$$

4.3 Monotone relations

We will now investigate support for special relationships that are axiomatized according to axiom (4.1). So assume this axiom schema for binary relations $\prec_1$ and $\prec_2$ and an uninterpreted function symbol f. Thus, whenever $x \prec_1 y$ holds, then for every set of auxiliary parameters $\overline{z}$, $\overline{u}$, we have $f(\overline{z}, x, \overline{u}) \prec_2 f(\overline{z}, y, \overline{u})$. To simplify notation, but without losing generality, we will assume that f is binary such that $\overline{z}$ is empty and $\overline{u}$ contains only one variable.

4.3.1 A ground decision procedure for monotone relations

As in the case for partial orders we obtain efficient support for deciding ground consequences of monotone relationships via a combination with the union-find data-structure used in congruence closure. We also obtain an incremental algorithm by considering the following two cases occurring when new facts are being asserted, and new terms are generated:

$q_1 \prec_1 q_2$**:** where q_1 and q_2 are root nodes in the union-find structure.

1. If $q_1 \prec_1 q_2 \in \mathit{Edges}$ we can assume that the congruence closure structure already knows about the fact, and we do not perform anything more.

2. If on the other hand $q_1 \not\prec_1 q_2 \in \mathit{Edges}$ the accumulated constraints are unsatisfiable and we notify this.
3. Finally, neither $q_1 \prec_1 q_2 \in \mathit{Edges}$, nor $q_1 \not\prec_1 q_2 \in \mathit{Edges}$, and we add the fresh edge $q_1 \prec_1 q_2$. For each $q \in \mathit{use}(q_1)$, $q' \in \mathit{use}(q_2)$, where $\mathit{canonical}(q)$, $L_{\mathcal{Q}}(q) = f(q_1, q_3)$, $\mathit{canonical}(q')$, $L_{\mathcal{Q}}(q') = f(q_2, q_4)$, add an edge $\mathit{find}(q) \prec_2 \mathit{find}(q')$. We check that there is not already an edge $\mathit{find}(q) \not\prec_2 \mathit{find}(q')$, otherwise the constraints are unsatisfiable.

$q_1 \not\prec_1 q_2, q_1 \prec_2 q_2, q_1 \not\prec_2 q_2$**:** where q_1 and q_2 are root nodes in the union-find structure. We add the edge $q_1 \not\prec_1 q_2$ (resp. $q_1 \prec_2 q_2$, $q_1 \not\prec_2 q_2$) to *Edges* checking that there is not already a contradictory edge. Notice that we do not have to close the congruence closure structure under the contrapositive of the special relations rule.

$f(q_1, q_2) \mapsto q$ **is inserted into** $L_{\mathcal{T}}$**:** First notice that the congruence closure algorithm maintains that whenever $f(q_1, q_2)$ is inserted into $L_{\mathcal{T}}$ both q_1 and q_2 are roots. We then take $Q_1' = \{\mathit{find}(q') \mid L_{\mathcal{Q}}(q') = f(q_1', q_2') \wedge q_1 \prec_1 q_1' \in \mathit{Edges}\}$ and add edges $q \prec_2 q'$ for each $q' \in Q_1'$. We also need to take $Q_2' = \{\mathit{find}(q') \mid L_{\mathcal{Q}}(q') = f(q_1', q_2') \wedge q_1' \prec_1 q_1 \in \mathit{Edges}\}$ and add edges $q' \prec_2 q$ for each $q' \in Q_2'$.

With Q being the number of different union-find nodes each operation described above has running time bounded by $\mathcal{O}(|Q|^2)$. The modular way in which the union-find structure is updated with new constraints enables independent support for several other special relationships $\prec_1', \prec_2', f' \ldots$, and works well with the incremental way that we will be maintaining constraints.

4.3.1.1 Correctness

The utilities for maintaining $\prec_1$, $\prec_2$ and f are clearly sound, as an examination of each step reveals. On the other hand, we establish their completeness by extracting a model from any non-contradictory union-find structure satisfying all asserted special relations.

Theorem 4.3.1 (Completeness) *Any consistent union-find structure saturated with respect to the special relations rules is satisfiable.*

Proof:

We construct a model satisfying the ground set of literals and all special relation axioms from the final non-contradictory state of the union-find structure. The model $\mathcal{A} = \langle A, \prec_1^{\mathcal{A}}, \prec_2^{\mathcal{A}}, f^{\mathcal{A}}, g^{\mathcal{A}}, h^{\mathcal{A}}, \ldots \rangle$ consists of

1. The domain A, which we identify with $\{q_0\} \cup \{\mathit{find}(q) \mid q \in Q\}$, where q_0 is a union-find node not already in Q.
2. The binary relations $\prec_1^{\mathcal{A}} \subseteq A \times A$ and $\prec_2^{\mathcal{A}} \subseteq A \times A$. We set $\prec_1^{\mathcal{A}} = \{(q, q') \mid q \prec_1 q' \in \mathit{Edges} \ \vee \ q = q_0 \ \vee \ q' = q_0\}$, and $\prec_2^{\mathcal{A}} = \{(q, q') \mid q \prec_2 q' \in \mathit{Edges} \ \vee \ q = q_0 \ \vee \ q' = q_0\}$.

3. The functions $f^{\mathcal{A}}$, $g^{\mathcal{A}}$, et.c.. We set $h^{\mathcal{A}}(q_1, \ldots, q_n) = \mathit{find}(L_{\mathcal{T}}(h(q_1, \ldots, q_n)))$ if $h(q_1, \ldots, q_n) \in \mathbf{dom}(L_{\mathcal{T}})$, otherwise $h^{\mathcal{A}}(q_1, \ldots, q_n) = q_0$.

To establish that the special relations are satisfied we will show that $f^{\mathcal{A}}$ is the appropriate interpretation of f such that whenever $q, r, s \in A$ and $(q, r) \in \prec_1^{\mathcal{A}}$, then $(f^{\mathcal{A}}(q, s), f^{\mathcal{A}}(r, s)) \in \prec_2^{\mathcal{A}}$.

For suppose to the contrary,

$$\mathcal{A} \models q \prec_1 r \ \wedge \ \neg(f(q, s) \prec_2 f(r, s)) \ .$$

Then it must be the case that $f(q, s), f(r, s) \in \mathbf{dom}(L_{\mathcal{T}})$. This implies that none of q, r or s equals q_0, as this was a fresh union-find node that could not have been found in the original union-find structure. Since $\mathcal{A} \models q \prec_1 r$ and q and r are different from q_0 it must be the case that $q \prec_1 r \in \mathit{Edges}$. The union-find structure may have been updated in two different ways:

1. $q' \prec_1 r'$ was inserted when both $f(q', s')$ and $f(r', s')$ were present in $L_{\mathcal{T}}$, where $\mathit{find}(q') = q$, $\mathit{find}(r') = r$ and $\mathit{find}(s') = s$ after all constraints have been processed. In this case the utilities for incrementally maintaining the monotonicity constraints would have added an $\prec_2$ edge between the finds of $L_{\mathcal{T}}(f(q', s'))$ and $L_{\mathcal{T}}(f(r', s'))$. The congruence closure algorithm maintains the invariants that $\mathit{find}(L_{\mathcal{T}}(f(q, s))) = \mathit{find}(L_{\mathcal{T}}(f(q', s')))$ (Invariant 3.3.8) and similarly for $f(r, s)$, thus ensuring that the $\prec_2$ edge connects precisely the nodes that were assumed not to be connected. A contradiction.
2. At least one of $f(q', s')$ or $f(r', s')$ were inserted after the $q' \prec_1 r'$ edge was established, where $\mathit{find}(q') = q$, $\mathit{find}(r') = r$ and $\mathit{find}(s') = s$ after all constraints have been processed. The incremental way special relations are maintained ensures to add an appropriate $\prec_2$ edge from the find of $L_{\mathcal{T}}(f(q', s'))$ to the find of $L_{\mathcal{T}}(f(r', s'))$. This also contradicts the assumption that $\mathcal{A} \models \neg(f(q, s) \prec_2 f(r, s))$ as the $\prec_2$ edges are always propagated to the roots of $L_{\mathcal{T}}(f(q', s'))$, $L_{\mathcal{T}}(f(r', s'))$.

■

4.3.2 Rigid S-unification

Suppose that we are given a set S of special relationships

$$\begin{aligned}
x \prec_{i_1} y &\rightarrow f_{j_1}(\ldots, x, \ldots) \prec_{k_1} f_{j_1}(\ldots, y, \ldots) \\
x \prec_{i_2} y &\rightarrow f_{j_2}(\ldots, x, \ldots) \prec_{k_2} f_{j_2}(\ldots, y, \ldots) \\
&\vdots \\
x \prec_{i_n} y &\rightarrow f_{j_n}(\ldots, x, \ldots) \prec_{k_n} f_{j_n}(\ldots, y, \ldots)
\end{aligned}$$

where $i_1, \ldots, i_n, k_1, \ldots, k_n$ range over some index set of binary relations and $j_1 \ldots, j_n$ are (not necessarily distinct) indices of function symbols. We extend rigid E-unification to constraints over S by defining *rigid S-unification.*

Definition 4.3.2 (Rigid S-unification) *A rigid S-unifier for the Horn-clause φ*

$$\bigwedge_i s_i = t_i \ \wedge \ u \prec_1 v \quad \rightarrow \quad w \prec_2 z \tag{4.8}$$

is a substitution θ from the free variables in φ to ground terms such that $\varphi\theta$ is valid in the theory of equality and S.

We establish that the rigid S-unification problem is decidable and in fact NP-complete. But, to convey the main ideas in a simplified way we first solve the rigid S-unification problem when S consists of a single monotonicity requirement

$$x \prec y \ \rightarrow \ f(\ldots, x, \ldots) \prec f(\ldots, y, \ldots) \tag{4.9}$$

for a fixed (uninterpreted) function f of arity $m + 1 + n$. The general case can then be handled by a slight extension of the arguments we give.

To nest a term t inside a number of applications of f define

Definition 4.3.3

$$f^{j+1}(t) \stackrel{\text{def}}{=} f(x_1, \ldots, x_m, f^j(t), y_1, \ldots, y_n) \quad \textit{where } x_i \textit{ and } y_i \textit{ are fresh variables}$$

$$f^0(t) \stackrel{\text{def}}{=} t$$

For example, for $m = 1, n = 2$

$$f^2(t) = f(x_1, f(x_2, t, y_1, y_2), y_3, y_4) \ .$$

Lemma 4.3.4 *Let E be a set of equalities, t and u be terms, and n be the number of sub-terms in E and terms t and u.*

Either there is a maximal $j \leq n$ such that

$$E \ \rightarrow \ u = f^j(t) \tag{4.10}$$

has a rigid E-unifier, or for all $m \geq 0$

$$E \ \rightarrow \ u = f^m(t) \tag{4.11}$$

has no rigid E-unifier.

Proof outline:

Suppose that there is a j and a rigid E-unifier for (4.10). We must establish that a $j \leq n$ can be chosen for this purpose. From Corollary 3.5.4 the rigid E-unifier can

be represented as an ordered set of pairs in the term-graphs of $E, u, f^j(t)$ and we can derive

$$\theta \ \wedge \ E \ \ \rightarrow \ \ u = f^j(t) \tag{4.12}$$

using congruence closure. So consider the partition $\mathcal{C}$ of the terms and sub-terms in θ, E, t, and u obtained by congruence closure with respect to the equalities E and θ. Divide the partition into two parts $\mathcal{C}_1$ and $\mathcal{C}_2$, where each class in $\mathcal{C}_1$ contains a term already a sub-term of E, t, or u, and the classes in $\mathcal{C}_2$ consists of terms not from E, t, or u. Suppose that $f^i(t)$ is not in $\mathcal{C}_1$, for some $0 < i \leq j$. Then it is because θ has mapped a variable x in E or u to a term containing the class $f^i(t)$. The substitution θ is consequently *pumped down* to replace the term equivalent to $f^i(t)$ by a term equivalent to t instead.

Consequently $f^i(t) \in \bigcup \mathcal{C}_1$, for all $i \leq j$. As there are at most n classes in $\mathcal{C}_1$, an index $j > n$ implies that some class in $\mathcal{C}_1$ is repeated. In this case we use the fact that the fresh auxiliary variables in $f^j(t)$ are all different to modify θ at will to pump j down below n.

▟

Theorem 4.3.5 *For a special relation given by (4.9) the S-unification problem for clauses of the form:*

$$\bigwedge_i s_i = t_i \ \wedge \ u \prec v \ \ \rightarrow \ \ w \prec z \tag{4.13}$$

is NP-complete.

Proof:

We first notice that (4.13) has a rigid S unifier θ if and only if there is a j, fresh binary function symbol h, and extension θ' of θ, that agrees with θ on the free variables in (4.13) such that

$$\left(\bigwedge_i s_i = t_i \ \ \rightarrow \ \ h(f^j(u), f^j(v)) = h(w, z) \right) \theta' \tag{4.14}$$

Thus, a rigid E-unifier θ' for (4.14) provides a rigid S-unifier for the original constraint (4.13). Lemma 4.3.4 provides an upper bound on the maximal number it makes sense to unfold w and z to match u and v, namely up to the number of sub-terms in the original S-unification problem. The lemma implies that further unfolding beyond this to test for solvability does not reveal anything new. Therefore, a sufficiently small j and a corresponding rigid E-unifier can be guessed and checked in polynomial time. This establishes that the special case of rigid S-unification is in NP.

To establish NP-hardness, notice that we can reduce the rigid E-unification

problem

$$\bigwedge_i s_i = t_i \quad \rightarrow \quad u = v \tag{4.15}$$

to the rigid S-unification problem

$$\bigwedge_i s_i = t_i \ \wedge \ x \prec x \ \rightarrow \quad u \prec v \tag{4.16}$$

where S is given by equation (4.9), where the monotone function f is not in (4.15). ⌟

To solve the general S-unification problem for more than a single monotonicity requirement one can perform a similar reduction as in Theorem 4.3.5 by non-deterministically examining one of the possible unfoldings leading from $\prec_1$ to $\prec_2$ which has length not exceeding the number of sub-terms in the original clause.

4.4 Summary

We gave ground decision support for selected special relations with the aim at combining these into the decision procedures. We also showed how to extract unifiers to close branches in a refutation search.

Chapter 5

Arithmetic

Constraints over relations with arithmetical operations appear in almost all verification conditions arising from simple sequential programs over reactive, real-time and hybrid systems. Even when hardware is modeled at an certain level of abstraction, arithmetical constraints become a natural part of the system model. Most of these arithmetical constraints are *linear*, in that multiplication is only used when at least one of the operands is a numeral. In the verification of hybrid systems, however, *non-linear* constraints appear naturally as a by-product of solving differential equations. We therefore aim here at building into a common framework decision procedures for arithmetic which (1) accomodates the frequently occurring linear arithmetical constraints efficiently, while (2) decides a reasonable fraction of constraints involving multiplication.

Since arithmetical constraints are so fundamental in system modeling and verification there is a vast literature on this subject already. The present exposition does not go into any impressive depth, but does offer an all-round treatment of decision procedures for linear and non-linear arithmetic. In particular, the fact that the Fourier-Motzkin procedure allows to extract equational constraints eagerly is not obvious from any of the references I am aware of, so we prove this for the purpose of fitting the linear solver into the combination of decision procedures. The solver for non-linear arithmetical constraints is furthermore guided using a sign-based abstraction domain to simplify polynomials and quickly detect redundant and inconsistent constraints.

5.1 Linear arithmetic

Linear arithmetic is the calculus obtained by including only terms of the form x_i, $a_i x_i$, $t+s$, $t-s$, and a_0, where $a_0, \ldots$ are rational constants $x_1, \ldots$ are rational variables, and s and t are linear arithmetical terms. Constraints are formed using the relation symbols $\neq$, $=$, $\leq$ and $<$. Linear arithmetical terms can be canonized by converting the terms into summation normal form:

$$a_0 + \sum_{i=1}^{n} a_i x_i$$

This assumes that the set of arithmetical variables $x_1, \ldots, x_n$ is ordered with respect to an aribitrary total ordering $\prec$. Assume for notational simplicity that this corresponds to the indexing, such that $x_1 \prec x_2 \prec \ldots \prec x_n$.

5.1.1 Equalities

A pure equality constraint is either vacuously true, unsatisfiable, or allows to eliminate one arithmetical variable by expressing it in terms of the others:

$$s = t \quad \leftrightarrow \quad 0 = t - s \quad \leftrightarrow \quad 0 = a_0 + \sum_{i=1}^{n} a_i x_i \quad \leftrightarrow \quad x_1 = -\frac{a_0}{a_1} + \sum_{i=2}^{n} -\frac{a_i}{a_1} x_i$$

By maintaining a fixed order of all terms and variables the resulting expression for x_1 is given uniquely.

Example: An example from the constraint programming literature is to prove that the sequence

$$x_{i+2} = |x_{i+1}| - x_i$$

starting from arbitrary initial values x_0 and x_1 is periodic and has period of length 9. This amounts to establishing unsatisfiability of

$$\bigwedge_{i=0}^{11} x_{i+2} = |x_{i+1}| - x_i \quad \wedge \quad \neg(x_0 = x_{10} \wedge x_1 = x_{11}) \ .$$

Expanding out the definition of $|x_1|$ once reduces $x_2 = |x_1| - x_0$ to $x_1 \geq 0 \ \wedge \ x_2 = x_1 - x_0 \ \vee \ x_1 < 0 \ \wedge \ x_2 = -x_1 - x_0$. By isolating x_0 in each disjunction reduces the original formula to

$$\begin{array}{rl} & x_1 \geq 0 \ \wedge \ \bigwedge_{i=1}^{11} x_{i+2} = |x_{i+1}| - x_i \quad \wedge \quad \neg(x_1 - x_2 = x_{10} \wedge x_1 = x_{11}) \\ \vee & x_1 < 0 \ \wedge \ \bigwedge_{i=1}^{11} x_{i+2} = |x_{i+1}| - x_i \quad \wedge \quad \neg(-x_1 - x_2 = x_{10} \wedge x_1 = x_{11}) \ . \end{array}$$

Nine more iterations of this expansion reduces the formula to false.

5.1.2 Inequalities

While the handling of linear equalities is completely standard in the Shostak-style integration of decision procedures, the question of how linear inequalities may be supported in an equational decision procedure integration has remained more open-ended.

5.1.2.1 Methods for integrating arithmetic

Shostak suggests in [Sho79] to use well established satisfiability checking methods, based on linear programming, such as Simplex [Dan62, Sch86, Chv83] or the SUP-INF method [Ble75, Sho77] to determine satisfiability of a set of linear arithmetic constraints, and in the affirmative case extract a model assigning each variable to a rational. Each arithmetical term

appearing in the combined constraints is then evaluated with respect to the assignment. Having associated all arithmetical terms with a rational constant we can invoke the remaining decision procedures on this instance assuming all they need to know about arithmetical terms are whether two terms are equal or not. The assignment may impose more equalities than implied by the original constraints. So this requires the arithmetical solver to be re-invoked should the satisfiability of the instance require two terms equated by the first assignment to be different. The same ideas carry over to linear constraints over integers and have been extended to a ground procedure with a *permutation* predicate in [SJ80] (see also [Mat81, Jaf81]).

Nelson in his Thesis [Nel81] presents an incremental procedure, which furthermore extracts implied equalities. It is unfortunate that this approach has not been more visible. Imbert and Hentenryck [IH93] elaborate with a similar perspective. Nelson's approach is formulated for the Nelon-Oppen combination of decision procedures, which does not target the extraction of equalities as substitutions that we require. We state here without proof that Nelson's approach can also be used to extract required equalities as substitutions by extending the proof we give for the Fourier-Motzkin elimination procedure to Nelson's Simplex tableau. Empirical data comparing Nelson's tableau and implementations of Fourier's algorithm for the integration of decision procedures would be useful.

There are specialized and efficient procedures for deciding satisfiability of a set of linear inequalities [Meg83] when all inequalities contain at most two variables. The approach extends [AS80, Nel82] as well as [Sho81] which actually attempts to generalize the method to handle three and more variables. Unfortunately, serious gaps in [Sho81] make an implementation of the ideas presented there very difficult if not impossible. The idea of looking at two variables per constraint can be traced back to [Pra77], while the connections with finding all pairs of shortest paths in a graph (via the Floyd-Warshall algorithm) should be obvious.

5.1.2.2 The Fourier-Motzkin variable elimination method

The approach suggested here is based on the classical Fourier-Motzkin variable elimination method, which gives a fully symbolic approach to testing satsifiability of linear constraints. This entails that implied equality constraints can be extracted and communicated with the other decision procedures. In NQTHM the Fourier-Motzkin procedure is referred to as *cross-multiplication* [BM88], but the presentation is to a great deal obscured by features specific to NQTHM. The Fourier-Motzkin procedure also forms the basis of theory integration in PVS and SVC [BC98]

The Fourier-Motzkin procedure eliminates a variable x_1 from a set of linear inequalities

$$a_{j0} + \sum_{i=1}^{n} a_{ji} x_i \ \geq 0, \qquad j = 1, \ldots, m$$

by rewriting these first to

$$\begin{aligned} x_1 &\geq a_{j0} + \sum_{i=2}^{n} a_{ji}x_i \quad j = 1, \ldots, m' \\ x_1 &\leq a_{j0} + \sum_{i=2}^{n} a_{ji}x_i \quad j = m'+1, \ldots, m'' \\ 0 &\leq a_{j0} + \sum_{i=2}^{n} a_{ji}x_i \quad j = m''+1, \ldots, m \end{aligned}$$

and then replace the first m'' inequalities by $m' \cdot (m'' - m')$ new inequalities, one for each $j = 1, \ldots, m'$ and $k = m'+1, \ldots, m''$:

$$a_{j0} + \sum_{i=2}^{n} a_{ji}x_i \geq a_{k0} + \sum_{i=2}^{n} a_{ki}x_i \ .$$

It is immediate that the new set of inequalities is satisfiable if and only if the old set is satisfiable. This approach also generalizes to strict inequality constraints. In fact the Fourier-Motzkin procedure is a quantifier elimination method for linear arithmetic. The elimination of x_1 above was precisely computing the quantifier-free equivalent to $\exists x_1 \ . \bigwedge_{j=1}^{m} a_{j0} + \sum_{i=1}^{n} a_{ji}x_i \ \geq 0$. It has been observed that the procedure has exponential time complexity [Sch86] and it is commonly percieved as far worse than the Simplex method. On the other hand, it enjoys some key algebraic properties that we will exploit. Further theoretical observations are made in [HLL90] and by Imbert in [SvH95].

5.1.2.3 An incremental equality-extracting Fourier-Motzkin procedure

The Fourier-Motzkin method as it stands does not allow to introduce more constraints involving a variable x_1 once it has been eliminated. Furthermore, it does not directly suggest which equality constraints are implied from a set of inequalities. For example, in

$$x \leq y + 2 \ \wedge \ y \leq z + 4 \ \wedge \ z + 6 \leq x$$

we would preferably infer $x = y + 2 = z + 6$. This will allow x and z to be replaced throughout other constraints and, for instance, establish verifiation conditions of the form

$$x \leq y + 2 \ \wedge \ y \leq z + 4 \ \wedge \ z + 6 \leq x \quad \rightarrow \quad f(x + y) = f(z + x + 4)$$

by detecting the inconsistency of

$$x \leq y + 2 \ \wedge \ y \leq z + 4 \ \wedge \ z + 6 \leq x \ \wedge \quad f(x + y) \neq f(z + x + 4)$$

via the simplified form

$$x = y + 2 \ \wedge \ z = y - 4 \ \wedge \ f(y + 2 + y) \neq f(y - 4 + y + 2 + 4)$$
$$\updownarrow$$
$$f(2 + 2y) \neq f(2 + 2y)$$

To support incremental addition of constraints we will not be able to eliminate constraints over variables that have been formally eliminated, but rely on the total ordering $\prec$ of variables to ensure termination. Linear inequalities are thus maintained in one of the forms:

$$x_j \ \geq \ a_0 + \sum_{i>j} a_i x_i \qquad x_j \ > \ a_0 + \sum_{i>j} a_i x_i$$
$$x_j \ \leq \ a_0 + \sum_{i>j} a_i x_i \qquad x_j \ < \ a_0 + \sum_{i>j} a_i x_i$$

where j is the smallest index with non-zero coefficient in the relevant inequality. We use $\mathcal{C}$ to refer to the collection of inequality constraints. Whenever a new inequality $t \leq s$ is added to the set of known inequalities we saturate the resulting set of inequalities with respect to the steps:

1. Isolate the variable with smallest non-zero coefficient in the inequality $t \leq s$ to obtain the equivalent inequality $x_j \leq u$ and add this to the set of known inequalities.
2. For each matching inequality $v \leq x_j$, resp. $w < x_j$ form the derived inequalities $v \leq u$ (resp. $w < u$).
3. Repeat step 1 with these newly derived inequalities.

This procedure terminates, since each step examines only inequalities whose variables have strictly higher indices. On the other hand, it may generate exponentially may inequalities requiring both exponential time and space.

We can use the approach to also derive all implied equalities by detecting $\leq$-loop-residues in the following way (we call this method a *loop-residue accumulation*): Whenever we add a non-strict inequality $s \leq t$, maintain the list

$$(x_{j1}, t_1), \ (x_{j2}, t_2), \ \ldots, (x_{jn}, t_n)$$

where

$$\begin{array}{lcl} s \leq t & \leftrightarrow & x_{j1} \leq t_1, \\ s_1 \leq x_{j1} & & \text{is used to match } x_{j1} \leq t_1 \\ s_1 \leq t_1 & \leftrightarrow & x_{j2} \leq t_2 \\ & \vdots & \end{array}$$

Then, if we derive the tight inequality

$$0 \leq 0$$

we may infer the triangular form

$$x_{j1} = t_1, \quad x_{j2} = t_2, \quad \ldots, \quad x_{jn} = t_n \ .$$

To see this inspect the last steps which from $s_n \leq x_{jn} \leq t_n$ forms $0 \leq t_n - s_n = 0$. In particular $t_n = s_n$ so $x_{jn} = t_n$. By replacing x_{jn} by t_n we may repeat the same eliminations of $x_{j(n-1)}$ up to x_{j1}. The triangular form is converted into a substitution θ and $\mathcal{C}$ is reduced to $\mathcal{C}\theta$.

We therefore have

Lemma 5.1.1 (Equational Soundness) *Any equality derived by loop-residue accumulation is an equality.*

On the other hand, this approach allows to infer all implied equalities:

Lemma 5.1.2 (Eager Equational Completeness) *Suppose that the set of inequalities $\mathcal{I}$ entails the equality $t = 0$, then either $\mathcal{I}$ is inconsistent or the substitution θ obtained from loop-residue accumulation on $\mathcal{I}$ satisfies $t\theta = 0$.*

Proof:

The proof is by induction on the number of variables in $\mathcal{I}$ of index greater or equal to the variables in t.

Assume therefore that $\mathcal{I} \models t = 0$. Thus both $\mathcal{I} \cup \{t < 0\}$ and $\mathcal{I} \cup \{t > 0\}$ are inconsistent. By completeness of the Fourier-Motzkin procedure we have that both augmentations of $\mathcal{I}$ lead to an inconsistency. Let $\mathcal{C}$ be the inequality constraints obtained from $\mathcal{I}$ by saturation, and let θ be the substitution obtained by accumulating loop residues. If $\mathcal{I}$ is inconsistent already we are done, otherwise augment $\mathcal{C}$ by $t\theta > 0$ arriving at an inconsistent inequality $r > 0$, where $r \in (-\infty..0]$. Separately we add $t\theta < 0$ to $\mathcal{C}$ to arrive at an equally inconsistent inequality.

If $t\theta$ is a constant it must be the case that $t\theta = 0$ for both $t\theta < 0$ and $t\theta > 0$ to be unsatisfiable. If $t\theta$ is not a constant we can rewrite the inequality $t\theta > 0$ as $x_1 > t_1$ (or $x_1 < t_1$), where x_1 is the variable with smallest index having non-zero coefficient. Symmetrically $t\theta < 0$ is written as $x_1 < t_1$.

We derive a contradiction from these inequalities by matching $x_1 > t_1$ with an inequality $s_1 \geq x_1$ (or $s_1 > x_1$) in $\mathcal{C}$, and $x_1 < t_1$ with an inequality $s_1' \leq x_1$ (or $s_1 < x_1$). Since $\mathcal{C}$ has been saturated with respect to its inequalities it furthermore contains all consequences of the combined constraint $s_1' \leq s_1$. On the other hand, both $t_1 < s_1$ and $s_1' < t_1$ are inconsistent. This can only be the case if $\mathcal{C}$ implies $s_1' = s_1$ and therefore $x_1 = s_1$. Now, the equality $s_1' = s_1$ involves only variables with indices higher than x_1, so the induction hypothesis implies $s_1'\theta = s_1\theta = x_1\theta$ contradicting the existence of the constraints $s_1 \leq x_1 \leq s_1'$ in $\mathcal{C}$.

∎

From Lemma 5.1.1 and 5.1.2 we now have all relevant ingredients to obtain the function *addConstraint* that accumulates arithmetical constraints incrementally while extracts all derived equalities.

The formulation chosen here allows us to infer that all intermediate inequalities can be turned into equalities. This generalizes the observation (Theorem 1) in [LHM93] *that an inequality* $\alpha_i x \leq \beta_i$ *is an implicit equality in a constraint set* $\mathcal{C}$ *iff Fourier's algorithm produces an inequality* $0 \leq 0$ *as a linear combination of constaints containing it.*

5.1.3 Disequalities

When variables range over rationals and reals it is not necessary to process disequalities $s \neq t$ other than checking $s\theta \neq t\theta$ for generated substitutions. In the congruence closure combination this reduces to checking that 0 never gets connected with a $\neq$ edge as we form the disequality constraint $0 \neq s - t$. This relies on the following property of linear arithmetic over the rationals:

Lemma 5.1.3 (Convexity) *For every* $t_1, \ldots, t_n$*, if* θ *is the substitution obtained from saturating a satisfiable set of inequalities* $\mathcal{I}$*, and* $\theta(t_i) \neq \theta(t_j)$*,* $i \neq j$*, then* $\mathcal{I}$ *is has a model where* $t_i \neq t_j$*, for* $i \neq j$*.*

5.1.4 Extracting models

Suppose that a set of constraints $\mathcal{C}$ contains the variables $x_1 \prec x_2 \prec \ldots \prec x_n$, and we wish to find an assignment of rationals to $x_1, \ldots, x_n$ satisfying $\mathcal{C}$. Such an assignment can be found by first collecting the set of inequalities of the form $l \leq x_n$, $x_n \leq u$[1] Since x_n is the variable with highest index l and u must necessarily be constants satisfying $l < u$. Any rational q between l and u can be legally assigned to x_n and we can repeat the procedure on $\mathcal{C}[x_n \mapsto q]$ to extract an assignment for x_{n-1} until all variables have been assigned a rational value.

The approach is also used for interfacing with other decision procedures by providing more general functions `SUP` and `INF`. The domain of both functions consists of a constraint set $\mathcal{C}$ and an arithmetical expression t, and the range of `SUP` is $\mathcal{Q} \cup \{\infty\}$, whereas the range of `INF` is $\mathcal{Q} \cup \{-\infty\}$. $\texttt{SUP}(\mathcal{C}, t)$ is computed from $(\mathcal{C}', \theta') = addConstraint(\mathcal{C}, \{x_{dummy} \leq t\})$ where x_{dummy} is a fresh variable whose index is higher than any variables contained in $\mathcal{C}$. In the result θ' is an identity substitution. If the updated constraint set $\mathcal{C}'$ contains an inequality $x_{dummy} \leq q$ (or $x_{dummy} < q$) [2] we set $\texttt{SUP}(\mathcal{C}, t) := q$ otherwise $\texttt{SUP}(\mathcal{C}, t) := +\infty$. $\texttt{INF}(\mathcal{C}, t)$ is computed in a dual way via $addConstraint(\mathcal{C}, \{x_{dummy} \geq t\})$.

Extensions of the Fourier-Motzkin procedure to include integer linear arithmetic [Pug91] provides the corresponding functionality for integers.

[1] In the absense of an inequality $l \leq x_n$ we set $l := -\infty$, similarly $u := +\infty$.

[2] q must necessarily be a rational constant, since the new variable has highest index.

5.1.5 Examples

Hardware modeling: A very good example for benchmarking the linear arithmetical decision procedures is the verification of the SRT division table presented in [CGZ96]. Procedures for linear arithmetic and records are required to close most branches. The main theorem, that the coefficient is guessed correctly at each cycle, is verified in less than 60 seconds[3].

The speed of verification does however depend on the order the axioms are listed. Splitting axioms in the wrong order creates linear constraints that the Fourier elimination method has obvious problems with. On the other hand, the 60 seconds here improve on the reported 3.5 hours using Matlab or the Mathematica based Analytica.

Program analysis: To check violations of array bounds, integer linear or linear arithmetic solvers can be used to resolve constraints from program analyzers. While the analyzer from [Diw98] has so far used the Omega package [Pug91] as its constraint solver for integer linear constraints, a collection of 100 sample data from different experiments gave precisely the same answers with the rational linear arithmetic as with integer linear arithmetic. This could be taken as a heuristic argument for using a rational linear arithmetic solver even for integer linear arithmetic constraints.

For reasons most likely connected to implementation and not theoretical limitations, our ML solver could handle at least one larger benchmark not handled by the Fortran based Omega package.

5.2 Non-linear arithmetic

This section describes an extension of the linear solver to the non-linear case, i.e., the case where variables can be multiplied to form non-linear multivariate polynomials.

Verification of non-linear, or symbolic hybrid systems produces verification conditions with non-linear polynomials. Small examples from [MS98] are listed in Table 5.1. Experiments with the commercial Redlog package were competitive for the first example, furthermore Redlog provides quantifier elimination for alternating quantifiers. Unfortunately, Redlog is a stand-alone tool and does not integrate smothely with solvers, and does for instance not handle division in inequalities, so it was not possible to use Redlog on examples 2 and 3. It should be noted that other highly optimized tools exist for checking satisfiability of non-linear inequalities, such as Numerica [HMD97]. Support for non-linear arithmetic from first principles can be found in state of the art verification systems, such as PVS, where the prelude includes well over 250 basic lemmas of non-linear arithmetic over the reals. These lemmas are all established automatically using the decision procedure described here.

It has been known since Tarski [Tar51] that the satisfiability problem for constraints over the real-closed field are decidable by quantifier elimination. Although *cylindric algebraic decomposition* [Col75, Hon92] can be used to perform quantifier elimination in doubly

[3] Allegedly Intel's SRT implementation for the Pentium processor contained bugs in the lookup table resulting in a 500 million dollar recall

No.	*Formula*	*sec.*
1.	$m \leq l + d \cdot r \ \wedge \ r < 0 \ \wedge \ x + t \leq d \ \wedge \ t > 0$ $\rightarrow \ m \leq l + r \cdot x + r \cdot t$	0.03
2.	$\left(\begin{array}{l} v_1 > 0 \ \wedge \ vr > 0 \ \wedge \ p \leq a \\ \wedge \quad x_2 + p/v_1 + i/vr \geq a/v_1 + a/vr \ \wedge \ p \geq i \end{array} \right)$ $\rightarrow \ a/v_1 + a/vr + v_1 \cdot t/v_1 \leq x_2 + t + p/v_1 + i/vr$	0.06
3.	$\left(\begin{array}{l} 2 \cdot (a-i)/vr + (a-i)/v_2 < x_1 \ \wedge \ v_2 > 0 \\ \wedge \quad vr > 0 \ \wedge \ p \geq i \ \wedge \ p \leq a \end{array} \right)$ $\rightarrow \ (a-i)/vr + (p-i)/vr + (a-i)/v_2 \leq x_1$	0.06

Table 5.1: Sample non-linear constraints

exponential space there are still several challenges in providing "practial" procedures for full elimination of quantifiers. One systematic approach is to use a *Gröbner basis solver* [Buc65] to simplify polynomials, though even computing a Gröbner basis can be costly. One can also add more ad hoc approaches such as simplifying non-linear constraints using a data-base of rewrite rules [DS97] to eliminate trivial redundancies.

The path taken here adapts the partial quantifier elimination procedure from [Wei97] to eliminate variables that occur with degree not greater than two, and couples this tightly with a sign-based abstraction domain and loosely with the linear-arithmetic solver. The equalities inferred by the linear solver are used to eliminate variables in the constraint solver for non-linear arithmetic. On the other hand, the non-linear solver infers polarities of multiplied terms that are propagated as inequality constraints to the linear solver. In this way we aim to exploit the best of both approaches: efficiency with expressibility.

5.2.1 A partial method for quantifier elimation

We are given the problem of deciding the satisfiability of a conjunction of inequalities between polynomials. Since the quantifier-free conjunction is satisfiable if and only if the existential closure is valid, methods from quantifier elimination (so-called projection) for polynomials become a natural tool for establishing satisfiability.

We will here review a partial method for eliminating quantifiers from multivariate polynomials whose variables occur with low degree. It eliminates a variable from a polynomial inequality by solving the variable to be eliminated symbolically. For example, in the polynomial $a \cdot x + b$, where a and b may contain variables different from x a formal solution for x in the equality $a \cdot x + b = 0$ is $x = -\frac{b}{a}$ subject to $a \neq 0$. In solving $a \cdot x + b > 0$ we introduce a symbolic parameter ϵ and get $x = -\frac{b}{a} + \epsilon$ subject to $a > 0$. To model arbitrary large positive and negative values for the eliminated variable formal symbols $\pm\infty$ are also added to the language. To allow substitution of expressions involving subterms of the form $\frac{b}{a}$, ϵ and $\pm\infty$ define:

Definition 5.2.1 ([LW93])

$$a \cdot (\pm\infty) + b = 0 \stackrel{def}{=} a = 0 \wedge b = 0 \tag{5.1}$$

$$a \cdot (\pm\infty) + b \geq 0 \stackrel{def}{=} a = 0 \wedge b \geq 0 \ \vee \ \pm a > 0 \tag{5.2}$$

$$a \cdot (\pm\infty) + b > 0 \stackrel{def}{=} a = 0 \wedge b > 0 \ \vee \ \pm a > 0 \tag{5.3}$$

$$a \cdot (\pm\infty) + b \neq 0 \stackrel{def}{=} a = 0 \wedge b \neq 0 \ \vee \ a \neq 0 \tag{5.4}$$

$$\tag{5.5}$$

$$a \cdot (c + \epsilon) + b = 0 \stackrel{def}{=} a = 0 \ \wedge \ b = 0 \tag{5.6}$$

$$a \cdot (c + \epsilon) + b \geq 0 \stackrel{def}{=} \left(\begin{array}{rl} & a \geq 0 \ \wedge \ a \cdot c + b \geq 0 \\ \vee & a < 0 \ \wedge \ a \cdot c + b > 0 \end{array} \right) \tag{5.7}$$

$$a \cdot (c + \epsilon) + b \neq 0 \stackrel{def}{=} a \neq 0 \ \vee \ b \neq 0 \tag{5.8}$$

$$a \cdot (c + \epsilon) + b > 0 \stackrel{def}{=} \left(\begin{array}{rl} & a > 0 \ \wedge \ a \cdot c + b \geq 0 \\ \vee & a \leq 0 \ \wedge \ a \cdot c + b > 0 \end{array} \right) \tag{5.9}$$

Under the the assumption $c \neq 0$, to substitute the formal division of two polynomials $\frac{d}{c}$ for x in a polynomial $p(x)$ one defines:

$$a \cdot \frac{d}{c} + b \ \rho \ 0 \stackrel{def}{=} a \cdot d + b \cdot c \ \rho \ 0 \qquad \rho \in \{=, \neq\} \tag{5.10}$$

$$a \cdot \frac{d}{c} + b \ \rho \ 0 \stackrel{def}{=} \left(\begin{array}{rl} & a \cdot d + b \cdot c \ \rho \ 0 \ \wedge \ c > 0 \\ \vee & 0 \ \rho \ a \cdot d + b \cdot c \ \wedge \ c < 0 \end{array} \right) \qquad \rho \in \{<, \leq\} \tag{5.11}$$

$$\tag{5.12}$$

$$a \cdot (\frac{d}{c} + \epsilon) + b \ \rho \ 0 \stackrel{def}{=} a \cdot (d + \epsilon) + b \cdot c \ \rho \ 0 \qquad \rho \in \{=, \neq\} \tag{5.13}$$

$$a \cdot (\frac{d}{c} + \epsilon) + b \ \rho \ 0 \stackrel{def}{=} \left(\begin{array}{rl} & a \cdot (d + \epsilon) + b \cdot c \ \rho \ 0 \ \wedge \ c > 0 \\ \vee & 0 \ \rho \ a \cdot (d + \epsilon) + b \cdot c \ \wedge \ c < 0 \end{array} \right) \qquad \rho \in \{<, \leq\} \tag{5.14}$$

It follows from [LW93] that

Theorem 5.2.2 *Let x occur linearly in the formula $\varphi(x)$:*

$$\bigwedge_{i \in I} a_i \cdot x + b_i = 0 \ \wedge \ \bigwedge_{j \in J} a_j \cdot x + b_j \neq 0 \ \wedge \ \bigwedge_{k \in K} a_k \cdot x + b_k \geq 0 \ \wedge \ \bigwedge_{l \in L} a_l \cdot x + b_l > 0$$

Then $\exists x\ .\ \varphi(x)$ *is equivalent to*

$$\bigvee_{i\in I\cup K} a_i \neq 0 \wedge \varphi(-\frac{b_i}{a_i}) \vee \bigvee_{i\in J\cup L} a_i \neq 0 \wedge \varphi(-\frac{b_i}{a_i}+\epsilon) \vee \varphi(-\infty)$$

Though the branching factor is linear in the number of literals containing x and the degree of the generated polynomials may increase, the method can be adapted to work for surprisingly many applications and is furthermore extended to second and third degree variables [Wei97, Wei94], and, more elaborately, to the general case.

Example: Elimination of x from the equality

$$\varphi(x,y): \quad x\cdot y - 5 = 0$$

produces the constraint

$$y \neq 0 \ \wedge\ \varphi(\tfrac{5}{y}, y) \quad \text{which simplifies to} \quad y \neq 0$$

and the substitution $[x \mapsto \frac{5}{y}]$. On the other hand, elimination of x from the equality

$$\varphi(x,y): \quad x\cdot y = 0$$

produces the constraint

$$y \neq 0 \ \wedge\ \varphi(0,y) \ \vee\ \varphi(-\infty, y) \qquad \text{which simplifies using (5.1) to}$$
$$y \neq 0 \ \vee\ y = 0$$

The disjunction corresponds to a *split* with two branches. The respective branches generate the substitutions:

$$\theta_1:\ [x \mapsto 0], \qquad \theta_2:\ [x \mapsto x_{-\infty}, y \mapsto 0]$$

where $x_{-\infty}$ is a fresh variable. The incremental solving allows (in principle, as the present implementation does not return substitutions) to establish goals such as:

$$x\cdot y - 5 = 0 \ \rightarrow\ f(x+2) = f(\frac{5}{y}+2)$$

and

$$x\cdot y = 0 \ \rightarrow\ f(2\cdot x^{100}) = f(0) \ \vee\ g(x\cdot y^2 + y\cdot 5 + 1) = g(1)$$

for arbitrary functions f and g.

In the next section we describe our approach to simplify polynomial inequalities and eliminate redundant branches generated by the quantifier elimination procedure.

5.2.2 Simplification using abstract interpretation

We propose the use of a simple sign-based abstract interpretation domain to simplify and maintain polynomials. Polynomial inequalities are partially evaluated and simplified using only information about the signs of variables. The sign-evaluation allows to often also reduce the degree of polynomials in polynomial inequalities appearing in the benchmarks we have encountered so far. Also, its effect on limiting branching factors is dramatic. Even small examples produce a large branching factor when the elimination procedure is used without the sign-based evaluation. Together with a tight integration with the solver for linear inequalities this comprises a handy tool for mixed linear and non-linear arithmetical constraints.

Besides a conjunction of polynomial inequalities the non-linear solver maintains a partial map from variables to signs. The sign of a variable x is one of the following constraints

$$x = 0, \quad x \neq 0, \quad x > 0, \quad x \geq 0, \quad x < 0, \quad x \leq 0, \quad ?$$

Signs are partially ordered with respect to implication such that since $x < 0 \rightarrow x \leq 0$ the sign $x < 0$ is preferred for $x \leq 0$ as the sign for x. The sign "?" is preferrably avoided as it imposes no constraints. The partial order is illustrated in Figure 5.1.

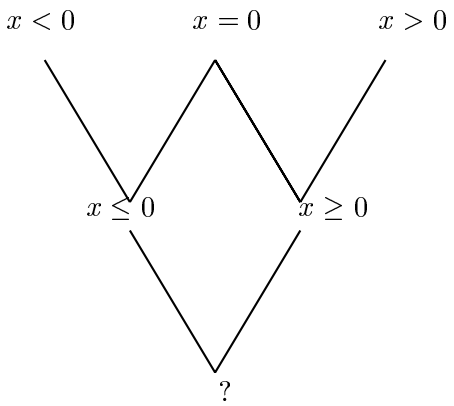


Figure 5.1: A Hasse diagram for the partial order of sign constraints

Signs for variables are first accumulated by inspecting the polynomials presented to the non-linear solver. For example, a polynomial inequality $x^3 > 0$ results in the sign constraint $x > 0$. Signs are evaluated using the obvious rules, such as

$$\begin{array}{lcl} (x > 0) \cdot (y < 0) & = & (x \cdot y < 0) \\ (x : ?)^2 & = & (x^2 \geq 0) \\ (x > 0) + (y < 0) & = & (x + y : ?) \end{array}$$

Since squaring and multiplication preserve more sign-information than addition we evaluate polynomials after multiplication has been distributed outwards as much as possible. Polynomial inequalities can then be simplified further based on the sign evaluation. For instance, if $p \neq 0$ is required, then each multiplicant that has a strictly non-zero sign ($<$, $\neq$, $>$) is eliminated. Naturally, sign-evaluation can do early detection of inconsistencies and simplify polynomials.

Example: Assume $x > 0$ and $y > 0$. We can then simplify

$$x \cdot y \cdot z + z^2 \cdot x \cdot y > 0$$

to the equivalent

$$z + z^2 > 0$$

because

$$\begin{aligned} & x \cdot y \cdot z + z^2 \cdot x \cdot y > 0 \\ \leftrightarrow \quad & x \cdot y \cdot (z + z^2) > 0 \\ \leftrightarrow \quad & y \cdot (z + z^2) > 0 \\ \leftrightarrow \quad & z + z^2 > 0 \end{aligned}$$

We do use methods that would split the last constraint to $z > 0 \ \vee \ z < -1$, but instead establish satisfiability directly by eliminating z.

5.2.3 Integration between linear and non-linear solvers

While asserted inequality constraints are initially passed to both the linear and non-linear solvers the best features from each are transferred to the other.

If possible, equalities derived in the linear solver are written in solved form $x = t$ where x is a variable that does not occur under multiplication in t. Whenever this is possible the substitution $[x \mapsto t]$ is besides being applied to the context in the linear solver also being applied to the non-linear solver. Since the linear solver is equationally complete (lemma 5.1.2) we obtain a full detection of implicitly fixed variables. This will therefore (slightly) generalize the features of the "naive" solver reported in [Col93] for Prolog III.

Polarities derived in the non-linear solver are conversely made visible to the linear solver to for instance make an early detection of the inconsistency

$$x > 0 \ \wedge \ y > 1 \ \wedge \ x \cdot y \leq -x - y$$

because the non-linear solver adds the constraint $x \cdot y > 0$ based on the polarities $x > 0 \ \wedge \ y > 0$. The resulting set of linear constraints are contradictory.

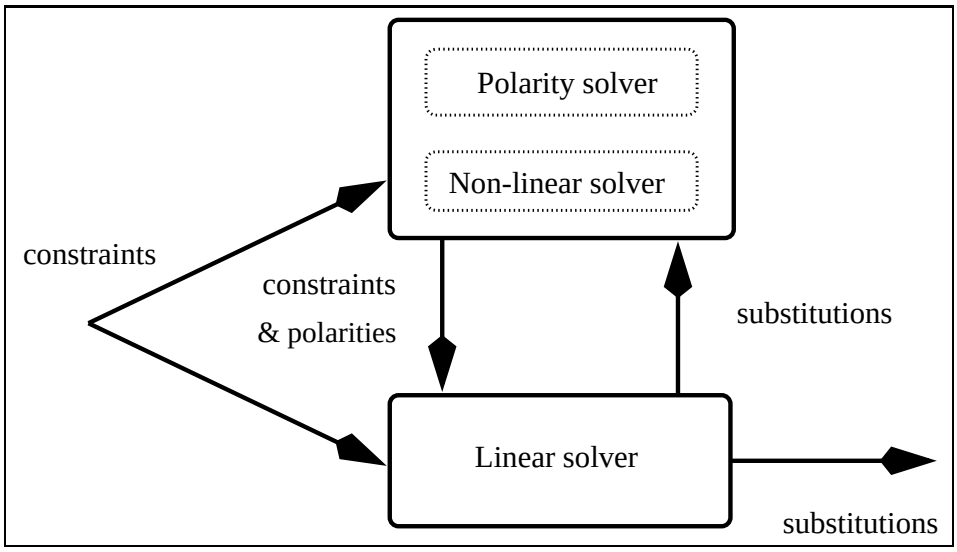


Figure 5.2: Integration between the linear and non-linear solvers

5.3 Summary

This chapter presented a combination of linear and non-linear solvers for arithmetical constraints. We used Fourier's method for elimination of variables to decide satisfiability for inequality constraints and also extract implied equalities as substitutions on the fly.

Chapter 6

Recursive and co-recursive data types

In this chapter we investigate decision procedures for recursive and co-recursive data types and how these can be integrated within the congruence closure-based decision procedure. The chapter falls into two parts.

In the first part, Section 6.1, we discuss theoretic properties of data types. Recursive data types are generated from a set of constructors and supplied with induction schemas ensuring *no junk* (the data type domain is the least set that can be obtained by applying constructors) and *no confusion* (elements are given by a unique sequence of applications of constructors). Recursive data types correspond to initial algebras (free term algebras). Co-recursive data types are conversely supplied with a co-induction schema, which ensures *maximal junk* (the elements of the data type domain is the largest set that can be obtained by applying the constructors), but still *no confusion* is required. Co-recursive data types correspond to final algebras. Co-recursive data types are essentially the infinite term trees in Prolog III [Col84]. In logic programming, so-called *sorted feature trees* [Smo92, NP93, Tre96] contain several similarities with co-recursive data types.

In the second part, Section 6.2, our goal is to show how constraints involving equations, disequations, inequations (the subterm relationship), and arithmetical constraints can all be integrated within the same combination. In particular, we present optimized algorithms for handling disequations, inequations, selectors, and unification of non-well-founded terms. When the domain of a data type is infinite we demonstrate how disequations can be tested for satisfiability in a processing-by-demand combination. The data-structures we use allow to handle inequations saving some redundant branching as compared to [Ven87, Tul94]. By treating selectors as uninterpreted within the data type theory, but interpreting them externally, we obtain a lazy approach to selector evaluation. This allows a solver-based decision procedure integration and has also demonstrated significant speedup on benchmarks using selectors. The constructor part of the theory of data types can then be dealt with using efficient Robinson-style unification algorithms. The subterm relation for well-founded data types is finally coupled with arithmetical constraints on the size of data types

in Section 6.2.6.

6.1 The theory of (co-)recursive data types

The material in this Section summarizes a number of facts about recursive and co-recursive data types. Several results may be derived as special cases from well-known general results. Other results are particular to first-order theories of data types.

6.1.1 Signatures for sorted data types

By a recursive *data type* τ we understand an implicitly defined sort characterized by a signature of the form

$$\langle \tau, S_1, \ldots, S_n, \Sigma \rangle$$

where $S_1, \ldots, S_n$ are different non-empty sorts and Σ is a finite set of constructors. Each constructor c in Σ has an associated arity:

$$c : T_1 \times \cdots \times T_{n_c} \to \tau$$

where $n_c \geq 0$, and each of the sorts T_i is taken from the list $\tau, S_1, \ldots, S_n$.[1]

With each constructor $c \in \Sigma$ we also associate a set of *selectors* $s^c_1, \ldots, s^c_{n_c}$, and a *tester* $\mathtt{is}c : \tau \to \mathcal{B}$.

Example: A signature for a domain of binary trees over a base sort S can be specified using

$$\langle \mathit{tree}, S, \mathit{node} : \mathit{tree} \times \mathit{tree} \to \mathit{tree}, \mathit{leaf} : S \to \mathit{tree} \rangle \ .$$

With *node* we associate the selectors *left* : *tree* $\to$ *tree* and *right* : *tree* $\to$ *tree* and tester is*node* : *tree* $\to \mathcal{B}$. With *leaf* we associate the selector *leaf-contents* : *tree* $\to S$ and the tester is*leaf* : *tree* $\to \mathcal{B}$.

Various characteristics of signatures lead to important special cases. We say that a signature is:

well-founded if there is a constructor that does not have τ in its domain.

linear if all constructors have τ occurring in at most one place.

singular if there is only a single constructor c of arity $\tau \times \tau \ldots \times \tau \to \tau$.

flat if τ does not appear in the domain of any constructor.

enumerative if each constructor has arity τ (does not take any arguments)

[1] Thus we limit ourselves to a very simple theory of data types. In particular, τ occurs only co-variantly in the domain of each constructor. data types with contra-variant dependencies have a much more involved model-theory.

a record if there is only one constructor and τ does not appear in the domain of that constructor.

Lists are a special case of linear data types. Lists are also a special case of queues, treated in chapter 8. The results in this chapter are therefore aimed exclusively at non-linear and flat data types. Singular data types have only trivial models, so these are ignored. By treating records and enumeration types as data types we can reuse the available tools developed here without having to duplicate effort on these types.

6.1.2 Canonical models

We will discuss two natural models for data types: recursive and co-recursive models. Since these are in a good sense dual to each-other we discuss them in the same Section.

6.1.2.1 Initial algebras

Assume we are given an interpretation I_0 for the sorts $S_1, \ldots, S_n$. The class of possible interpretations for τ that we shall consider are all extensions of I_0. The initial algebra I_{init} is the extension of I_0 such that for any other extension I there is a homomorphism $h : I_{init} \to I$. This definition (which is a special case of the more general one from [Bir35]) is well founded as we have

Proposition 6.1.1 ([Grä79]Corollary 24.1, Theorem 24.2) *If I_{init} exists it is unique up to isomorphism.*

We give a proof for our special case, as the same techniques are used for final co-algebras.

Proof:

Let I and J be initial such that $I\lceil S_i = J\lceil S_i = I_0$, $i = 1, \ldots, n$. Then by assumption homomorphisms $h : I \to J$ and $g : J \to I$ exist, and for every term $t(x_1, \ldots, x_m)$ with $x_j \in S_i$,

$$\begin{aligned} g \circ h(t(x_1, \ldots, x_m))^I &= g \circ h(t^I(x_1^I, \ldots, x_m^I)) \\ &= g(t^J(x_1^J, \ldots, x_m^J)) \\ &= t^I(x_1^I, \ldots, x_m^I) \\ &= (t(x_1, \ldots, x_m))^I \end{aligned}$$

So $g \circ h$ is an isomorphism on the term universe of $\langle \tau, \overline{S}, \Sigma \rangle$. ∎

To construct an initial model I_{init}, and later construct dual final models, we define a *τ-tree*:

Definition 6.1.2 (τ-trees) *A τ-tree consists of a $\langle T, \lambda, sort \rangle$, where*

1. *T is a non-empty prefix-closed subset of $\{1 \ldots \max\{arity(c) \mid c \in \Sigma\}\}^*$,*
2. *$\lambda : T \to \Sigma \cup I_0(S_1) \cup \ldots I_0(S_n)$ is a labeling, and*

3. *sort* $: T \to \{\tau, S_1, \ldots, S_n\}$ *labels the nodes by sorts,*

such that

1. *If* $sort(t) = \tau$, *then* $\lambda(t) = c$ *for some* $c \in \Sigma$, *and if* c *has arity* $T_1 \times \ldots \times T_{n_c} \to \tau$, *then* $t1, \ldots, tn_c \in T$, *but* $ti \notin T$ *for* $i > n_c$, *and* $sort(tj) = T_j$ *for* $j = 1, \ldots, n_c$.

2. *If* $sort(t) = S_j$ *for some auxiliary sort* S_j, *then* $\lambda(t) \in I_0(S_j)$, *and* $ti \notin T$ *for all* i.

Finite τ-trees *are the trees where* T *has finite cardinality, and* infinite τ-trees *are the trees where* T *is not restricted cardinal-wise. The set of* rational trees *is obtained by taking the* τ*-trees that have only a finite number of different sub-trees.*

Theorem 6.1.3 I_{init} *exists.*

Proof:

The term-algebra over $I_0(S_1), \ldots, I_0(S_n)$, where every distinct term over Σ corresponds to unique elements is isomorphic to the set of finite τ-trees. Let I be another interpretation. We construct $h : I_{init} \to I$ by recursion as a union $\bigcup_{i<\omega} h_i$. For $i = 0$ we define h_0 as the identity map on the range of I_0. Inductively assume that h_i is given, and let $c(t_1, \ldots, t_n)$ be a term where $t_1, \ldots, t_n$ are terms of depth at most i. Then

$$I(t_j) = h_i(I_{init}(t_j)) = h_i(t_j) \qquad j = 1, \ldots, n$$

We now set

$$h_{i+1}I_{init}(c(t_1, \ldots, t_n)) = h_{i+1}(c(t_1, \ldots, t_n)) = I(c(t_1, \ldots, t_n)) = I(c(h_i(t_1), \ldots, h_i(t_n)) \ .$$

∎

The interpretation of testers is now uniquely given by the axioms

$$\forall x \in \tau \ . \ \forall c \in \Sigma \ . \ \mathtt{is}c(x) \ \leftrightarrow \ \exists y \in \mathbf{dom}(c) \ . \ x = c(y) \ . \tag{6.1}$$

While the interpretation of constructors and testers is unique (up to isomorphism) we admit any extension of I_{init} satisfying the selector axioms:

$$\forall c \in \Sigma \ . \ \forall (y_1, \ldots, y_n) \in \mathbf{dom}(c) \ . \ s_i^c(c(y_1, \ldots, y_n)) = y_i \ . \tag{6.2}$$

This leaves the interpretation of selectors under-specified when the selector does not match the constructor associated with the term where the selector is applied. Hodges [Hod93], for instance insists that $s_i^c(c'(y_1, \ldots, y_n)) = c'(y_1, \ldots, y_n)$ when $c \neq c'$ to obtain a unique interpretation of selectors. This works only in an unsorted setting, however. Treinen [Tre91] maps non-matching selector applications to a fixed element $\perp_S$ for each sort S. If selectors are guaranteed only to be applied to terms, whose top-most constructor matches the selector,

we may get rid of the selectors in an initial phase using transformations of the following form:

$$\mathcal{C}[s_i^c(t)] \qquad \mapsto \qquad \mathcal{C}[x_i] \ \wedge \ t = c(x_1, \ldots, x_i, \ldots, x_n) \ . \tag{6.3}$$

where $x_1, \ldots, x_n$ are fresh (existentially quantified) variables. We are then back at the pure constructor theory. This transformation is only sound if t could only have been generated as an application of c. For instance

$$\mathit{left}(\mathit{leaf}(x)) \quad = \quad y \tag{6.4}$$

is satisfiable if *leaf* is a total function. But the transformation from (6.3) produces the unsatisfiable

$$\exists x_1, x_2 \ . \ x_1 = y \ \wedge \ \mathit{node}(x_1, x_2) = \mathit{leaf}(x) \ .$$

This situation actually arises in [Sho84], where the solver for data types is unsound and returns false when solving (6.4).

Alternatively, we can characterize the initial algebras with the axiomatization $\mathcal{I}$ in Figure 6.1.

$\mathcal{F}(\tau) \subseteq \tau$	(introduction)
$\forall X \subseteq \tau. \, \mathcal{F}(X) \subseteq X \ \rightarrow \ \tau \subseteq X$	(induction)
$\forall c_i, c_j \in \Sigma, \forall y_1 \in \mathbf{dom}(c_i), y_2 \in \mathbf{dom}(c_j) \ . \ c_i(y_1) = c_j(y_2) \ \rightarrow \ i = j \wedge y_1 = y_2$	(no confusion)
$\forall c \in \Sigma \ . \ \mathtt{is}c(x) \ \leftrightarrow \ \exists y \in \mathbf{dom}(c) \ . \ x = c(y)$	(tester)
$\forall c \in \Sigma, \forall (y_1, \ldots, y_n) \in \mathbf{dom}(c) \ . \ s_i^c(c(y_1, \ldots, y_n)) = y_i$	(selector)

Figure 6.1: Initial algebra axiomatization $\mathcal{I}$

To state these axioms we use the predicate transformer $\mathcal{F}$, defined:

$$\mathcal{F}(S) \quad \stackrel{\text{def}}{=} \quad \{c(y_1, \ldots, y_n) \mid c \in \Sigma, \ (y_1, \ldots, y_n) \in \mathbf{dom}(c)\lceil S\}$$

where $\mathbf{dom}(c)\lceil S$ is the domain of c where occurrences of τ are restricted to be included in the set S. The analogy with taking the post-condition from a set of states can be helpful to keep in mind.

Example: For the data type of trees we have:

$$\mathcal{F}(X) = \{\mathit{node}(x, y) \mid x, y \in X\} \cup \{\mathit{leaf}(x) \mid x \in S\}$$

So for instance

$$\mathcal{F}(X) \subseteq Y \quad \leftrightarrow \quad \forall x, y \in X, s \in S \;.\; node(x, y) \in Y \wedge \; leaf(s) \in Y$$

and

$$X \subseteq \mathcal{F}(Y) \quad \leftrightarrow \quad \forall x \in X \;.\; \exists y, z \in Y, s \in S \;.\; x = node(y, z) \;\vee\; x = leaf(s)$$

Returning to the axioms in Figure 6.1,

Proposition 6.1.4 *The introduction, induction, and no confusion axioms determine τ up to isomorphism.*

Proof:

From the induction axiom we have

$$\tau \subseteq \bigcap\{X \mid \mathcal{F}(X) \subseteq X\}$$

From the introduction axiom we have the converse

$$\bigcap\{X \mid \mathcal{F}(X) \subseteq X\} \subseteq \tau$$

In summary

$$\tau \;=\; \bigcap\{X \mid \mathcal{F}(X) \subseteq X\} \;=\; \bigcup_{\alpha<\omega} \mathcal{F}^{\alpha}(\emptyset) \;=\; \mu X \;.\; \mathcal{F}(X) \;.$$

In words, τ is the least set obtained by applying the constructors finitely many times to elements from the base sorts $S_1, \ldots, S_n$. Together with the (no confusion) axioms we conclude that τ coincides with the free term-algebra. ∎

6.1.2.2 Final co-algebras

We now investigate decision procedures for the case where data types are interpreted as final co-algebras.

An interpretation I_{final} is a final co-algebra in a class K of interpretations (which are as before all extensions of I_0) if for any I in K there is a homomorphism $h : I \to I_{final}$. The class K of interpretations we here have in mind are the *strongly extensional* models [Acz88]. Strongly extensional models are those where identity coincides with the largest bisimulation [Mil89].

Two elements $a, b \in \mathbf{range}(I)$ are bisimilar if there is a binary relation $R \subseteq \mathbf{range}(I) \times \mathbf{range}(I)$, such that $R \setminus (I(\tau) \times I(\tau)) = diag(I(S_1)) \cup \ldots \cup diag(I(S_n))$, and $(a, b) \in R$ and for every $a, a' \in I(\tau)$, $(a, a') \in R$ iff there are $\overline{b}, \overline{b}' \in \mathbf{range}(I)$, $c \in \Sigma$ such that $a = I(c(\overline{b}))$, $a' = I(c(\overline{b}'))$, $(b_i, b_i') \in R$, $\quad i = 1, \ldots, arity(c)$. The largest bisimulation is as usual the union of all bisimulations. Alternatively we can appeal to the Anti Foundation Axioms (AFA) to factor out bisimilar models. The same proof as for Proposition 6.1.1 gives

Proposition 6.1.5 *If I_{final} exists it is unique up to isomorphism.*

To construct I_{final} we will now use the domain of infinite τ-trees. It is straight forward to verify that equality on infinite τ-terms is a maximal bisimulation. Intuitively, two different τ-trees differ on a finite path, establishing that there can be no bisimulation between them. More formally,

Lemma 6.1.6 *Equality on infinite τ-trees is a maximal bisimulation.*

Proof:

Take any bisimulation relation R on τ-trees and suppose that

$$(\langle T_1, \lambda_1, sort_1\rangle, \langle T_2, \lambda_2, sort_2\rangle) \in R.$$

We prove by induction on the length of strings in T_1 and T_2, that they must coincide. The base case requires to establish that $\epsilon \in T_1$ iff $\epsilon \in T_2$, which is the case as both sets are non-empty and prefix-closed, $\lambda_1(\epsilon) = \lambda_2(\epsilon)$, and $sort_1(\epsilon) = sort_2(\epsilon)$, which follows by unfolding the condition on R once. As the induction hypothesis suppose that T_1 and T_2 have the same strings of length less than i and that $sort_1$ and $sort_2$ as well as λ_1 and λ_2 coincide on all strings of length less than i. Now take any string s of length $i-1$ (prefix closure of T_1, T_2 makes sure this is not a restriction) and define for $k = 1, 2$

$$T_k^s \stackrel{\text{def}}{=} \{t \in \{1 \ldots \max\{arity(c) \mid c \in \Sigma\}\}^* \mid st \in T_k\} \tag{6.5}$$

$$\lambda_k^s(t) \stackrel{\text{def}}{=} \lambda_k(st) \tag{6.6}$$

$$sort_k^s(t) \stackrel{\text{def}}{=} sort_k(st) \tag{6.7}$$

Now either $sort_1^s(\epsilon) = S_j$ for some sort S_j or $sort_1^s(\epsilon) = \tau$. In the first case $T_1^s = T_2^s = \{\epsilon\}$, and $\lambda_1^s = \lambda_2^i = [\epsilon \mapsto s]$ for some $s \in I_0(S_j)$. In the second case the conditions on R require that there is a $c \in \Sigma$ such that $\lambda_1^s(\epsilon) = \lambda_2^s(\epsilon) = c$ and for each $i = 1, \ldots, arity(c)$, $(\langle T_1^{si}, \lambda_1^{si}, sort_1^{si}\rangle, \langle T_2^{si}, \lambda_2^{si}, sort_2^{si}\rangle) \in R$. ∎

We will now fix the interpretation I_{final} as the one that maps every data type term t to the corresponding (isomorphic) τ-tree. Despite the naming I_{final}, we have yet to establish whether it is indeed a final co-algebra. This will not be the case when the models may contain unnecessary junk. To avoid this, we restrict K further to those interpretations where *domain closure* holds.

Definition 6.1.7 (Domain closure) *Domain closure holds for I in K if for every $a \in I(\tau)$ there are $c \in \Sigma$ and $\overline{b} \in$ **range**(I) such that $a = I(c(\overline{b}))$. In other words we require the interpretations in K to satisfy*

$$\tau \subseteq \mathcal{F}(\tau) \ .$$

Notice that domain closure holds for the construction we gave for I_{final}.

Theorem 6.1.8 *Let the class of K interpretations satisfy domain closure and $I_{final} \in K$, then I_{final} is a final co-algebra in K.*

Proof:

Take an arbitrary interpretation I in K. As I satisfies domain closure we can for each element in $I(\tau)$ fix an arbitrary one-step unfolding (there is actually precisely one by the (no-confusion) axiom). A one-step unfolding of $a \in I(t)$ chooses a $c \in \Sigma$ and $\overline{b} \in \mathbf{range}(I)$ such that $a = I(c(\overline{b}))$. We can now associate each element in $I(\tau)$ with a τ-tree by building it in stages based on the transitive unfolding obtained by applying the fixed one-step unfoldings. ∎

To capture finality axiomatically we can use the *solution lemma*, which states that all (consistent) sets of equations have (unique) solutions. The solution lemma and its relations to non-well-founded set-theory are discussed in [Acz88]. Stated using infinitary connectives and index sets I and J, it reads

$$\forall x : J \to \tau \;.\; \exists! y : I \to \tau \;.\; \bigwedge_{i \in I} y(i) = t_i(x, y)$$

where each $t_i(x, y)$ is a term over variables $x(j), j \in J$, $y(i), i \in I$, and there is no chain $i_1, i_2, \ldots, i_k, \ldots \in I$, such that for each pair $(a, b) \in \{(i_1, i_2), \ldots, (i_k, i_{k+1}), \ldots\}$, the term t_a is $y(b)$. This condition ensures that every variable $y(i)$ is (eventually) defined in terms of some term which is either of the form $x(j)$ or uses a constructor. Notice that this schema includes in K trees that are not rational.

We can therefore capture the final co-algebra axiomatization by the axioms in Figure 6.2 among strongly extensional models. Dual to induction, which implies domain closure, the solution lemma implies the principle of introduction $\mathcal{F}(\tau) \subseteq \tau$.

$\tau \subseteq \mathcal{F}(\tau)$	(domain closure)
$\forall x : J \to \tau \;.\; \exists! y : I \to \tau \;.\; \bigwedge_{i \in I} y(i) = t_i(x, y)$	(solution lemma)
$\forall c_i, c_j \in \Sigma, \forall y_1 \in \mathbf{dom}(c_i), y_2 \in \mathbf{dom}(c_j) \;.$ $c_i(y_1) = c_j(y_2) \;\to\; i = j \wedge y_1 = y_2$	(no confusion)
$\forall c \in \Sigma \;.\; \mathtt{is}c(x) \;\leftrightarrow\; \exists y \in \mathbf{dom}(c) \;.\; x = c(y)$	(tester)
$\forall c \in \Sigma, \forall (y_1, \ldots, y_n) \in \mathbf{dom}(c) \;.\; s_i^c(c(y_1, \ldots, y_n)) = y_i$	(selector)

Figure 6.2: Final co-algebra axiomatization $\mathcal{C}$

As an alternative to finality among extensional interpretations K, we can capture I_{final}

by asserting the principle of co-induction:

$$\forall X \subseteq K \;.\; X \subseteq \mathcal{F}(X) \;\rightarrow\; X \subseteq \tau \;.$$

The sort τ is constrained as the greatest fix-point to $\mathcal{F}$ as the co-induction principle asserts

$$\bigcup\{X \subseteq K \mid X \subseteq \mathcal{F}(X)\} \subseteq \tau$$

and the domain closure condition ensures the converse

$$\tau \subseteq \bigcup\{X \subseteq K \mid X \subseteq \mathcal{F}(X)\},$$

so in summary

$$\tau = \bigcup\{X \subseteq K \mid X \subseteq \mathcal{F}(X)\} = \nu X \subseteq K \;.\; \mathcal{F}(X) \;.$$

We can relate this to the constructed I_{final} by showing that it is a maximal fix-point and any other fix-point comes with an injection to I_{final}. A proof of this observation can be modeled directly after Theorem 14.1 in [BM96] page 198 where it is formulated for streams assuming AFA. The explicit use of K in the co-induction principle is also pervasive in [Pau93, Pau97]. Here, co-induction is embedded in HOL using encoding from first principles. For instance, for each base type α, K is the type α `node set set`, and terms are built from primitive operations $\otimes_D$, $\oplus_D$, for forming products and sums over non-well-founded structures.

One can naturally bypass the entire discussion of finality by modeling co-recursive data types directly using τ-trees as the basic notion. This has been done in [Fef96] in the case of streams.

6.1.3 Mixed data types

We have presented the sorted signatures for (co-)recursive data types for simplicity with only one data type attached. Consider now example 6.1.3.

Example: Mutual recursive definitions of trees and forests:

$$\left\langle \begin{array}{lll} S, & & \\ \tau_{tree}, & node : S \times \tau_{forest} \;\rightarrow\; \tau_{tree}, & branch : \tau_{tree} \;\rightarrow\; \tau_{tree} \\ \tau_{forest}, & nil : \tau_{forest}, \;\; cons : \tau_{tree} \times \tau_{forest} \;\rightarrow\; \tau_{forest} & \end{array} \right\rangle$$

In the case where τ_{tree} and τ_{forest} are both interpreted a recursive data types or both interpreted as co-recursive data types it does not take much effort to extend all definitions to support such mutually recursively defined types.

The more subtle question is to provide meaningful interpretations and support for a mixture of recursive and co-recursive data types. For instance, if we insist that the domain of τ_{tree} may include infinitely long branches, but that all forests should be finite one should be able to constrain τ_{tree} as a co-recursive data type and τ_{forest} as a recursive data type. In

general terms assume we have a structure

$$\langle \tau_c, \tau_r, S_1, \ldots, S_n, \Sigma_r, \Sigma_c \rangle$$

where Σ_r and Σ_c are sets of disjoint constructors. The constructors $f \in \Sigma_r$ have arity $T_1 \times \ldots \times T_{n_f} \to \tau_r$, $f \in \Sigma_c$ have arity $T_1 \times \ldots \times T_{n_f} \to \tau_c$ where $T_1, \ldots$ range over $\tau_c, \tau_r, S_1, \ldots, S_n$. We wish to obtain a recursive interpretation of τ_r and co-recursive interpretation of τ_c assuming well-foundedness (Σ_r contains a constructor that does not have τ_r in its domain).

Alternating τ-trees provide a way to obtain such an interpretation.

Definition 6.1.9 *An* alternating τ-tree *is a possibly infinite τ-tree $\langle T, \lambda, sort \rangle$ over sorts $\tau_c, \tau_r, S_1, \ldots, S_n$ such that there is no infinite subset $\{w_1, w_2, w_3, \ldots\}$ of T where $\forall i \exists j \,.\, w_{i+1} = w_i j$ and $\forall i \,.\, sort(w_i) = \tau_r$.*

Since in each case T is finitely branching, König's lemma implies that this requirement is equivalent to excluding infinite terms over τ_r.

Alternating τ-trees are not necessarily the only meaningful model. In [BS98] general conditions on structures including recursive and co-recursive data types are studied in order to achieve meaningful combinations and integrated decision procedures.

6.1.4 Equational theories

A ground equational formula is built exclusively from boolean combinations of equalities. A first-order equational formula φ is built from ground equational formulas by adding first-order quantification.

6.1.4.1 Recursive data types

The induction schema from Figure 6.1 is the only non-equational axiom for inductive data types. It implies two sets of equational axioms, namely domain closure, as well as that no term is a proper subterm of itself. The latter has to be formulated using an infinite supply of equational axioms, one for each term over Σ together with auxiliary variables. In summary we obtain the equational axiomatization in Figure 6.3.

6.1.4.2 Co-recursive data types

The axiomatization of co-recursive data types already contain axioms for domain closure. The finite instances of the solution lemma looks remarkably dual to the no-cycles condition. Thus, we state the corresponding equational axiomatization for co-recursive data types in Figure 6.4.

6.1.5 Beyond equational theories

While a second-order system allows to define derived relations, such as the subterm relation, a pure first-order system needs to introduce these separately. Hence, for the subterm relation

$\mathcal{F}(\tau) = \tau$	(fix-point)
$\forall x \in \tau, \forall \overline{y}. x \neq t(x, \overline{y})$	(no-cycles)
$\forall c_i, c_j \in \Sigma, \forall y_1 \in \mathbf{dom}(c_i), y_2 \in \mathbf{dom}(c_j)\ .$ $c_i(y_1) = c_j(y_2) \ \rightarrow\ i = j \wedge y_1 = y_2$	(no confusion)
$\forall c \in \Sigma\ .\ \mathtt{is}c(x) \ \leftrightarrow\ \exists y \in \mathbf{dom}(c)\ .\ x = c(y)$	(tester)
$\forall c \in \Sigma, \forall (y_1, \ldots, y_n) \in \mathbf{dom}(c)\ .\ s_i^c(c(y_1, \ldots, y_n)) = y_i$	(selector)

Figure 6.3: Equational system $\mathcal{I}_E$ for recursive data types

$\mathcal{F}(\tau) = \tau$	(fix-point)
$\forall \overline{x} \in \tau, \exists! \overline{y}.\ \bigwedge_i y_i = t_i(\overline{x}, \overline{y})$	(unique solutions)
$\forall c_i, c_j \in \Sigma, \forall y_1 \in \mathbf{dom}(c_i), y_2 \in \mathbf{dom}(c_j)\ .$ $c_i(y_1) = c_j(y_2) \ \rightarrow\ i = j \wedge y_1 = y_2$	(no confusion)
$\forall c \in \Sigma\ .\ \mathtt{is}c(x) \ \leftrightarrow\ \exists y \in \mathbf{dom}(c)\ .\ x = c(y)$	(tester)
$\forall c \in \Sigma, \forall (y_1, \ldots, y_n) \in \mathbf{dom}(c)\ .\ s_i^c(c(y_1, \ldots, y_n)) = y_i$	(selector)

Figure 6.4: Equational system $\mathcal{C}_E$ for co-recursive data types

$s \preceq t$, which holds iff s is a subterm of t is relevant for standard termination arguments of recursive programs. The single axiom-schema *subterm* encodes this relation.

A ground decision procedure integration is presented in Section 6.2.5.

6.1.6 First-order equational decision methods

6.1.6.1 Encodings into S2S

Suppose that each sort S_i can be encoded in an enumerable domain. We can then reduce decision problems for the first-order theory of constructors (but without selectors) to wS2S (the *weak* monadic second-order logic of two successors) for recursive data types respectively full S2S for co-recursive data types. This connection is perhaps not surprising. For instance [KS97] presents an encoding of recursive data types via wS2S and guided tree-automata. It has however not been possible to find a definite reference to this connection, so we discuss it in some depth here. The added value of using S2S is that the encoding also

$$\forall c \in \Sigma, \forall x, \forall y \in \mathbf{dom}(c)\ .$$
$$x \preceq c(y_1, \ldots, y_n) \ \leftrightarrow\ x = c(y_1, \ldots, y_n) \ \vee\ \bigvee_i x \preceq y_i \qquad \text{(subterm)}$$

Figure 6.5: subterm relation axiom schema

allows quantification over positions. The lost value includes problems in encoding under-specified selectors, the apparent need for full S2S for co-recursive data types (it is believed much more intractable than wS2S in practice [Kla98]), and impossible to extend the S2S-based representation to handle subterm relations using the same translation (the first-order theory with subterm relations is undecidable).

An encoding of terms using unary predicates is sketched below. The analogy with I_{init} and I_{final} should be kept in mind as we here essentially use binary tress to encode trees of arbitrary, but bounded branching. The distinction between the initial and final models is reflected in the type of quantifiers admitted.

1. Constructors. Assume that predicates $P_1, \ldots, P_n$ encode $t_1, \ldots, t_n$. Then P encodes $c_i(t_1, \ldots, t_{n_i})$, where c_i is the i'th function symbol in Σ with arity n, if the following conditions are satisfied:
   (a) P is downwards closed: $\forall x.P(x\mathtt{L}) \vee P(x\mathtt{R}) \rightarrow P(x)$.
   (b) $P(\epsilon)$.
   (c) The left branch departing ϵ has length i and does not split: $P(\mathtt{L}^i)$ but $\neg P(\mathtt{L}^{i+1})$ and $\neg P(\mathtt{L}^j\mathtt{R})$ for $1 \leq j \leq i$.
   (d) The right branch departing ϵ has length n: $P(\mathtt{R}^n)$ but $\neg P(\mathtt{R}^{n+1})$.
   (e) The j'th split on the right branch contains the j'th subterm: $\forall x.P(x\mathtt{R}^j\mathtt{L}) \ \leftrightarrow\ P_j(x)$ for $1 \leq i \leq n$.
2. To check that a predicate encodes a well-formed term we introduce the abbreviation

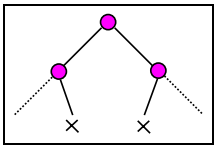
encodes-term(P), which is defined with:

$$\mathit{isPath}(x, P) \stackrel{\text{def}}{=} \forall y \geq x \;.\; \neg P(y\mathtt{L}) \vee \neg P(y\mathtt{R})$$

We use *isPath* to encode values from the data type domains S_i.

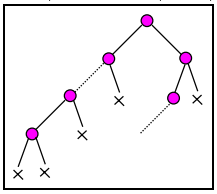
$$\mathit{sort}_\tau(x, P) \stackrel{\text{def}}{=} P(x\mathtt{L}) \wedge P(x\mathtt{R}) \wedge \neg P(x\mathtt{LR}) \wedge \neg P(x\mathtt{RL})$$

This constrains P to branch both left and right, but not followed by a zig-zag. It is illustrated to the right.



$$\begin{aligned}\mathit{sort}_{S_i}(x, P) \stackrel{\text{def}}{=}\ & P(x\mathtt{L}^i) \wedge \neg P(x\mathtt{L}^{i+1}) \wedge \bigwedge_{1 \leq j \leq i} \neg P(x\mathtt{L}^j\mathtt{R}) \\ \wedge\ & \mathit{isPath}(P, x\mathtt{RL}) \wedge P(x\mathtt{RL}) \wedge \neg P(x\mathtt{R}^2)\end{aligned}$$

This constrains P to branch straight left with a path of length i, and right in a path following a zig-zag movement. It is illustrated to the right.



$$\begin{aligned}\mathit{isGoodRoot}_i(x, P) \stackrel{\text{def}}{=}\ & P(x\mathtt{L}^i) \wedge \neg P(x\mathtt{L}^{i+1}) \wedge \bigwedge_{1 \leq j \leq i} \neg P(x\mathtt{L}^j\mathtt{R}) \\ \wedge\ & P(x\mathtt{R}^{n_i+1}) \wedge \neg P(x\mathtt{R}^{n_i+2}) \\ \wedge\ & \bigwedge_{1 \leq j \leq n_i} \mathit{sort}_{T_j}(x\mathtt{R}^{j+1}\mathtt{L}, P)\end{aligned}$$

Here c_i has arity $T_1 \times \cdots \times T_{n_i} \to \tau$.

$$\begin{aligned}\mathit{encodes\text{-}term}(P) \stackrel{\text{def}}{=}\ & \mathit{sort}_\tau(\epsilon, P) \\ \wedge\ & \forall x \;.\; \mathit{sort}_\tau(x, P) \;\to\; \bigvee_{c_i \in \Sigma} \mathit{isGoodRoot}_i(x, P) \\ \wedge\ & \forall x \;.\; P(x\mathtt{L}) \vee P(x\mathtt{R}) \to P(x)\end{aligned}$$

3. Testers are expanded according to the tester axioms.

4. Equalities of terms are now encoded as predicate equivalence (i.e., set equality).

5. First-order quantification is encoded as second-order quantification over unary predicates (sets) relativized to *encodes-term*. For recursive data types quantification is relativized to *finite* sets. This forces all terms to be finite. This can be accomplished

directly by using wS2S or by the predicate $\mathit{finite}(P)$, where

$$\mathit{finite}(P) \quad \stackrel{\text{def}}{=} \quad \neg(\exists R \,.\, R \subseteq P \ \wedge \ \mathit{escape}(R))$$

$$\mathit{escape}(R) \quad \stackrel{\text{def}}{=} \quad R \neq \emptyset \ \wedge \ \forall x \in R \exists y \in R \,.\, x < y$$

The characterization corresponds to a a weak form of König's lemma (for trees of branching degree 2). For co-recursive data types, quantification is unrestricted. Finite as well as infinite terms are admitted. Full S2S is required to represent quantification of infinite sets.

The freedom to relativize variables in S2S produces as a side-effect a decision procedure for first-order equational theory of mixed data types. We here have to relativize variables to those which do not contain an *escape* sequence of recursive constructors.

6.1.6.2 First-order quantifier elimination

An early quantifier elimination procedure for free term algebras with commutativity axioms can be found in [Mal71] (the original paper in Russian is from 1961). Maher [Mah88a] gives quantifier elimination procedure for recursive and co-recursive data types. Although presented in an unsorted setting it can be extended in a straight-forward way to multi-sorted data types [Mah88b]. While only equational axiomatizations are discussed, categoricity of the second-order axiomatizations $\mathcal{I}$ and $\mathcal{C}$ implies:

Corollary 6.1.10 *For every first-order formula φ, where every atomic formula is an equality between τ-terms without selectors:*

$$\mathcal{I} \models \varphi \quad \mathit{iff} \quad \mathcal{I}_E \models \varphi \quad \mathit{iff} \quad \mathcal{I}_E \vdash \varphi$$

and

$$\mathcal{C} \models \varphi \quad \mathit{iff} \quad \mathcal{C}_E \models \varphi \quad \mathit{iff} \quad \mathcal{C}_E \vdash \varphi \ .$$

Unfortunately Maher's decidability results do not extend to selectors when their interpretation is left under-specified. It is for instance straight forward to simulate binary predicates with a selector applied to a non-matching binary constructor. This allows to reconstruct two-counter machines and other Turing-complete devices. Rackoff proves [Rac75] and Vorobyov [Vor96] reproves that the first-order theories of recursive and co-recursive data types are non-elementary in the sense of Kalmar, i.e., cannot be decided within time bounded by a k-story exponential function for any fixed k. Both the quantifier elimination procedure and the embedding into S2S provide a comparable upper bound.

6.1.7 Related theories of data types

The theories of feature trees [Smo92] are related to the co-recursive data types discussed here. Sub-feature relationships are for instance studied in [MNT98]. Features do not have

the same arity restrictions that the data types have here. This makes sub-feature constraints harder (complete for PSPACE, they correspond to automaton simulations) than the NP-complete subterm relations studied here.

6.2 Decision procedure integration for data types

We will here develop procedures that can be used to verify claims like

```
type sexpr == CONS :: car : sexpr, cdr : sexpr | NIL

value x,y,z : sexpr
```

```
NIL <= CONS(NIL,NIL)
x <= y /\ y <= x --> x = y
x < y /\ y <= x --> false
NIL < CONS(NIL,NIL)
x < CONS(x,y)
x <= y --> x < CONS(y,z)
CONS(x,y) = CONS(y,z) --> x = z
car(x) = NIL /\ x = NIL --> car(car(x)) = NIL
length(CONS(NIL,NIL)) = 3
length(A) > length(B) --> !(A <= B)
```

all in neglible time with the same integration of decision procedures.

Without much added effort we also obtain procedures for automatically verifying claims for co-recursive data types, such as

```
cotype sexpr == CONS :: car : sexpr, cdr : sexpr | NIL

value x,y : sexpr
```

```
CONS(x,x) = x /\ CONS(y,y) = y --> x = y
CONS(x,y) = x /\ CONS(y,x) = x --> x = y
```

Due to STeP's focus on reactive systems the less trivial examples that these decision procedures have been exposed to have involved only records. For instance, a possible encoding of the version of the SRT lookup table presented in [RSS96] requires more attention to how record projections are handled. A preliminary version of record projection decision procedures based on Shostak's suggestions [Sho84] required 4 minutes to verify the main claim. With the lazy evaluation of the projection (selector) operations we present here it is verified in 10-30 seconds depending on how the theorem is presented.

The required machinery is being developed in the rest of this chapter.

6.2.1 τ-automata

The union-find structure used in the congruence closure induces a structure much like a top-down deterministic tree automaton, by taking Q as states and successor function $\delta = \mathit{children} \circ L_{\mathcal{Q}} : Q \to Q^*$. In general we can not assume that $L_{\mathcal{Q}}$ is always up to date with the union-find structure. In the revised definition of δ below, we therefore apply *find* to each child of $L_{\mathcal{Q}}(q)$ to anticipate a later update of $L_{\mathcal{Q}}$. By labeling each state q by the head function symbol in $L_{\mathcal{Q}}(q)$ we also obtain a way to access terms associated with the states.

When interpreting a specific data type τ with constructors $\Sigma \subseteq \mathcal{F}$ we will however use a modified successor function, which only produces successors for a state q, when it is labeled by a function symbol in Σ. Thus,

Definition 6.2.1 (τ-automaton) *Given the union-find structure with terms described in Section 3.2 and data type τ with constructors Σ, the τ-automaton is a tuple*

$$\mathcal{A} : \langle Q,\ \delta : Q \to Q^*,\ \lambda : Q \to \mathcal{F} \rangle$$

such that

$$\lambda(q) \stackrel{\text{def}}{=} \textbf{let}\ \ (f, \overline{q}) \ =\ L_{\mathcal{Q}}(q)\ \textbf{in}\ \ f$$

$$\delta(q) \stackrel{\text{def}}{=} \textbf{if}\ \ \lambda(q) \in \Sigma\ \textbf{then}\ \ \mathit{map}\ \mathit{find}\ (\mathit{children}(L_{\mathcal{Q}}(q)))\ \textbf{else}\ \ \langle\rangle$$

Since we have just defined τ-automata we will sneak in two auxiliary definitions associated with these automata. These concern paths in τ-automata and reachability.

Definition 6.2.2 (Paths: π) *A path π is a sequence of positive integers. The evaluation of state q on path π is written $\pi(q)$ and defined via:*

$$\begin{array}{lcl} \epsilon(q) & = & q \\ (i \cdot \pi)(q) & = & \pi(\delta(q)_i) \end{array}$$

where $\delta(q)_i$ is the i'th projection of $\delta(q)$ defined (arbitrarily) as q if $|\delta(q)| < i$. A path π is well formed *on q if π is ϵ or $\pi = i \cdot \pi'$, $|\delta(q)| \geq i$ and π' is well formed on $\delta(q)_i$. Paths are partially ordered by the string* prefix *relation.*

Definition 6.2.3 (Reachability) *Let q_1, q_2 be states in Q, then*

$$q_1 \trianglelefteq q_2 \quad \mathit{iff} \quad \mathit{there\ is\ a\ path}\ \pi\ \mathit{such\ that}\ q_2 = \pi(q_1)$$

6.2.2 Unification using τ-automata

Since terms are represented by the union-find node that corresponds to the top most sub term we can unify a pair of terms based on a Robinson-style unification algorithm [BS93] for

the pair of corresponding union-find nodes. More generally, given a set of pairs of union-find nodes $\mathcal{E}$ define *unify* in Figure 6.6.

$unify(\mathcal{E}) = unifyPairs(Id, \mathcal{E})$ where

$unifyPairs(\sigma, \emptyset) =$ **return** σ
$unifyPairs(\sigma, \{q_1 \stackrel{?}{=} q_2\} \cup \mathcal{E}) =$
  let
    $q_1' = (q_1)\sigma$ and $q_2' = (q_2)\sigma$
  in
    if $q_1' = q_2'$ **then** $unifyPairs(\sigma, \mathcal{E})$ **else**
    if $\lambda(q_1') \notin \Sigma$ **then** $unifyPairs(\sigma \cdot [q_1' \mapsto q_2'], \mathcal{E})$ **else**
    if $\lambda(q_2') \notin \Sigma$ **then** $unifyPairs(\sigma \cdot [q_2' \mapsto q_1'], \mathcal{E})$ **else**
    if $\lambda(q_1') = \lambda(q_2')$ **then** $unifyPairs(\sigma \cdot [q_1' \mapsto q_2'],$
      $\mathcal{E} \cup \{\delta(q_1')_i \stackrel{?}{=} \delta(q_2')_i \mid i \leq arity(L_{\mathcal{Q}}(q_1'))\})$
    else return FAIL

Figure 6.6: Unification using τ-automata

The result of *unify* is either FAIL, in which case the input terms do not unify, or a substitution σ mapping union-find nodes to union-find nodes. The restriction of σ where domain nodes are labeled by variables (i.e., whose head function symbols are not in Σ) induces a most general unifier. The easiest way to see this is perhaps by viewing the present algorithm as a refinement of Robinson's unification algorithm.

Operations associated with the substitution σ can be implemented using a union-find data-structure. We then obtain an almost linear-time unification algorithm as noted in [BN98]. Zhang [Zha92] gives a slightly more efficient "shell-nut" data-structure that works as a lazy union-find structure and solves the union-find problem for unification in constant time. Every step eliminates one state or discharges an equality. The entire unification process can therefore be implemented to run in time linearly in $\Sigma_{q \in Q} \max(1, |\delta(q)|)$ (using the Shell-Nut data-structure). In [JK90] it is left open whether rational trees could be unified in linear time, but the shell-nut procedure does precisely that.

6.2.3 Integration with congruence closure

Shostak [Sho84] proposes a solver for a special theory of S-expressions (convex S-expressions, where the axiom `x = CONS(CAR(x),CDR(x))` holds). With some goodwill it can be extended to other data types. However fundamental to this approach selectors like `CAR` and `CDR` are treated as interpreted symbols and may therefore not become part of a solved form.

We will use the unification algorithm from Figure 6.6 to solve equalities for recursive as

well as co-recursive data types. By separating out the treatment of selectors and testers we will be able to make some theoretical observations on differences in decision complexities, and also be able to obtain a solver that works for a maximally flexible interpretation of the selectors.

6.2.3.1 Recursive data types

Presented with an equality constraint $t = s$, where s and t are terms over the recursive data type τ, we can invoke the unification algorithm from Figure 6.6 on the pair of union-find nodes $\{q_s \stackrel{?}{=} q_t\}$. In case of failure s and t differ on a common position with incompatible constructors. The equality $t = s$ is then unsatisfiable. In case of success, the unification algorithm returns a substitution $[q_1 \mapsto q'_1, \ldots, q_n \mapsto q'_n]$. We can perform the occurs check á posterior in linear time using a topological sorting algorithm or using Tarjan's algorithm for finding strongly connected components in a graph. Tarjan's algorithm produces a partition of the states Q. We then check that each partition is a singleton set, without a looping state transition. This check can naturally be interleaved with the generation of strongly connected components and unification. This gives essentially the occurs check approach of [RP89]. Alternatively one can use the linear-time algorithm from [DST80] to perform the congruence closure of $\mathcal{A}$. This algorithm terminates if the graph contains a cycle. Note that the graph induced by $\mathcal{A}$ and the unifier contains a cycle if and only if the occurs check is violated.

Provided the occurs check is not violated the substitution $[q_1 \mapsto q'_1, \ldots, q_n \mapsto q'_n]$ from the unification algorithm is equivalent to a solved form

$$\bigwedge_{i=1}^{m} x_i = t_i$$

where $m \leq n$ and none of the x_i occur free in the t_i. As unification does not produce new states, no new terms need to be presented to the congruence closure before it can process the set of solved equalities $[q_1 \mapsto q'_1, \ldots, q_n \mapsto q'_n]$ by merging q_i and q'_i for $i = 1, \ldots, n$. Notice how the use of directed merge, which sets the find of q_i to that of q'_i, is consistent with the fact that if q_i is labeled by a constructor it coincides with the constructor labeling q'_i.

6.2.3.2 Co-recursive data types

The occurs check is not required for co-recursive data types. Instead the result of unification can produce bisimilar nodes that are not merged. For instance take the constraint

$$x = node(x, x) \ \wedge \ y = node(y, y) \ .$$

Before taking the equalities into account the associated union-find structure would allocate four nodes, q_1, q_2, q_3, q_4, where $L_{\mathcal{Q}}(q_1) = x$, $L_{\mathcal{Q}}(q_2) = node(q_1, q_1)$, $L_{\mathcal{Q}}(q_3) = y$, $L_{\mathcal{Q}}(q_4) = node(q_3, q_3)$. Asserting the equality $x = node(x, x)$ requires to unify q_1 and q_2 resulting

in the substitution $[q_1 \mapsto q_2]$. Separately the equality $y = node(y, y)$ causes merging q_3 with q_4. The resulting union-find structure now has $find(q_1) = q_2$, $find(q_3) = q_4$, $L_{\mathcal{Q}}(q_2) = node(q_2, q_2)$, and $L_{\mathcal{Q}}(q_4) = node(q_4, q_4)$. In the associated τ-automaton $\mathcal{A}$, q_2 and q_4 are different but bisimilar states, so therefore represent the same element in any model. To merge q_2 and q_4 (in other words ensure that terms are all in canonical form) we need to close $\mathcal{A}$ according to the maximal bisimulation relation satisfying (the Myhill-Nerode principle):

$$q \equiv q' \qquad \text{iff} \qquad \lambda(q) = \lambda(q') \ \wedge \ \forall i \leq |\delta(q)| \ . \ \delta(q)_i \equiv \delta(q')_i$$

Efficient algorithms for partition refinement [PT87] can minimize τ-automata in time bounded by $\mathcal{O}(n\log(n))$, where n is the size of Q. In an early paper Oppen [Opp80b] gives an $\mathcal{O}(n\log^2(n))$ bound based on other algorithms.

6.2.3.3 Satisfiability of equations and disequalities

The two refinements of the unification algorithm above give efficient procedures for deciding satisfiability of conjunctions of equalities and disequalities over recursive and co-recursive data types. Given a conjunction L of equalities $t = s$ and disequalities $u \neq v$ where all terms range over a data type τ, we can perform the steps of the algorithm in Figure 6.7.

1. Produce a union-find structure by applying *canonize* to each term in L.
2. Extract the τ-automaton $\mathcal{A}$ from the union-find structure.
3. Form the set $\mathcal{E} : \{q_s \stackrel{?}{=} q_t \mid s = t \text{ in } L\}$ and apply *unify* on $\mathcal{E}$.
4. In the case of recursive data types a linear-time congruence closure algorithm is sufficient to do occurs check and collapse nodes that must be equal. In the case of co-recursive data types, automaton minimization in $\mathcal{O}(n\log(n))$ suffices in order to collapse states that must be interpreted equally.
5. If the unification or minimization merges two nodes that are associated with a disequality or different constructors the original set L is unsatisfiable.

Figure 6.7: Algorithm for checking consistency of equalities and disequalities

When τ is non-singular, and not flat with all parameter sorts S_i being finite domain it is simple to generate infinitely many different terms of type τ. For instance, if τ is a well-founded recursive data type we can choose an assignment of "fresh" terms to root-nodes that are not labeled by constructors (the first non-constructor node is labeled by a term of size $|\{find(q) \mid q \in Q\}| + 1$, the second by a term twice the size, et.c., This entails that the algorithm in Figure 6.7 is complete for well-founded recursive data types. When the data type is an enumeration type, however, it is easy to reduce the graph k-coloring problem to the satisfiability problem by representing nodes in a given graph by different variables ranging over a data type of k elements and asserting disequalities corresponding to edges. The graph 3-coloring problem is NP-complete, so this leaves little hope for obtaining

efficient algorithms for the enumeration case. The procedure in Figure 6.7 therefore only serves as a partial consistency check when splitting is not employed.

The algorithm can also be used directly to solve the decision problems studied in [Col84]. The theoretical running time of our algorithm seems however better ($\mathcal{O}(n\log(n))$ as opposed to at least $\mathcal{O}(n^2)$, if not $\mathcal{O}(n^3)$[2]).

6.2.4 Selectors and testers

The efficient algorithmic results do not carry over to constraints including testers or selectors. Consider for example the data type

$$\langle \tau,\ T : \tau,\ F : \tau,\ cons : \tau \times \tau \to \tau \rangle \tag{6.8}$$

with selectors $car : \tau \to \tau$, $cdr : \tau \to \tau$ and testers $\mathsf{is}T$, $\mathsf{is}F$ and $\mathsf{is}cons$. Then given an instance $\varphi : \bigwedge_i (l_i \vee k_i \vee m_i)$ of 3-SAT where l_i, k_i, m_i are literals over the alphabet $\{x_1, \ldots, x_n\}$, we introduce fresh variables $\hat{x}_i$ and $\overline{\hat{x}}_i$ for the positive and negative literals respectively. Now φ is satisfiable if and only if

$$\begin{array}{ll} & \bigwedge_i cons(cons(F,F),F) \neq cons(cons(\hat{l}_i,\hat{k}_i),\hat{m}_i) \\ \wedge & \bigwedge_{x\in\mathcal{V}} \hat{x} \neq \hat{\overline{x}} \\ \wedge & \bigwedge_{x\in\mathcal{V}} \neg \mathsf{is}cons(\hat{x}) \wedge \neg \mathsf{is}cons(\hat{\overline{x}}) \end{array}$$

is satisfiable, if and only if

$$\begin{array}{ll} & \bigwedge_i cons(cons(F,F),F) \neq cons(cons(\hat{l}_i,\hat{k}_i),\hat{m}_i) \\ \wedge & \bigwedge_{x\in\mathcal{V}} \hat{x} \neq \hat{\overline{x}} \\ \wedge & \bigwedge_{x\in\mathcal{V}} cons(car(\hat{x}),cdr(\hat{x})) \neq \hat{x} \wedge cons(car(\hat{\overline{x}}),cdr(\hat{\overline{x}})) \neq \hat{\overline{x}} \end{array}$$

is satisfiable.

Nelson and Oppen [NO78] noticed that when S has infinite cardinality, then equalities and disequalities over

$$\langle \tau,\ S,\ atom : S \to \tau,\ cons : \tau \times \tau \to \tau \rangle \tag{6.9}$$

with selectors $car : \tau \to \tau$, $cdr : \tau \to \tau$ and testers $\mathsf{is}atom$, and $\mathsf{is}cons$, can be decided in time $\mathcal{O}(n^2)$. The complexity for these domains is also obtained using the present integration with congruence closure. Selectors are evaluated using the canonizer σ, which eliminates pairs of matching selectors and constructors.

To handle the general case we propose the approach in Figure 6.8. It suggests to delay interpretation of selectors as a last resort. For example, consider Shostak's approach when solving

$$\#1\ x_1 = 1\ \wedge\ \#1\ x_2 = 1\ \wedge\ \ldots\ \wedge\ \#1\ x_{100} = 1\ \wedge\ \#1\ x_{100} \neq 1$$

[2] That article does not provide a precise running time analysis

where x_i are tuples of length 100. This will first solve #1 $x_1 = 1$, which reduces to $x_1 = (1, y_2, \ldots, y_{100})$, introducing 99 fresh variables. The solution for x_1 may be propagated to other constraints before the second equality can be processed, introducing another 99 fresh variables. In the end, 9900 fresh variables are introduced before the contradiction is detected.

1. Whenever the association $[q \mapsto s(q')]$ is inserted into $L_{\mathcal{Q}}$, where s is a selector, we record the selector redex $s(q')$ provided $\lambda(q') \notin \Sigma$ (that is, if q' is not labeled by a constructor).

   To localize data type reasoning in the data type solver, this recording is provided by the canonizer σ, which takes $s(q')$ and attempts to simplify it if q' labels a matching constructor. As a side-effect it notices if q' was not a constructor.

2. The data type reasoner now has the option on splitting for each selector redex $s(q)$ introducing the splits $q = c_1(x^1_{new}), \ldots, q = c_n(x^n_{new})$ for each data type constructor $c_1, \ldots, c_n$ provided with fresh variables as arguments.

Figure 6.8: Algorithm for checking consistency in the presence of selectors.

We proceed with a somewhat involved argument for the completeness of this approach and introduce the following notation:

Definition 6.2.4 (Redex closure) *The automaton $\mathcal{A}$ is* closed under selector/constructor redexes *iff for every q in $\mathcal{A}$ if use(q) contains a node q', where $L_{\mathcal{Q}}(q') = s^c_i(q)$ then $\lambda(q) \in \Sigma$. Furthermore if $\lambda(q) = c$, then $\delta(q)_i = q'$, which means that the selector applied according to its definition.*

To witness the difference between states q_1 and q_2 in a τ-automaton closed under selector/constructor redexes we introduce the notion of a *state differentiator*.

Definition 6.2.5 (State differentiator) *A* state differentiator *for states q_1 and q_2 in automaton $\mathcal{A}$ is a pair (Π, T), where*

- *$\mathcal{A}$ is closed under selector/constructor redexes.*
- *Π is a set of integer sequences $\{\pi_0, \pi_1, \pi_2, \ldots\}$.*
- *$T \subseteq \Pi \times Q \times Q$ is a relation satisfying.*
  1. *$T(\pi_0, q_1, q_2)$*
  2. *For every path π and states q_1, q_2: $T(\pi, q_1, q_2)$ iff*
     $\lambda(\pi(q_1)) \neq \lambda(\pi(q_2))$ and
     for every prefix π' of π, and nodes q^1 and q^2, if $L_{\mathcal{T}}(s(q^1)) = \pi'(q_1)$, $L_{\mathcal{T}}(s(q^2)) = \pi'(q_2)$, for some selector s, then there is a $\pi'' \in \Pi$ such that $T(\pi'', q^1, q^2)$.

We can then introduce a congruence relation $\simeq$ on states:

Definition 6.2.6 (Congruence with selectors: $\simeq$) *States $q_1 \simeq q_2$ iff they have no differentiator (Π, T).*

An automaton is reduced if $\simeq$ partitions Q into singletons:

Definition 6.2.7 (Reduced automata) *The automaton $\mathcal{A}$ is* reduced *if:*

$$\textit{for every } q_1, q_2 \in Q, \textit{ if } q_1 \neq q_2 \textit{ then } q_1 \not\simeq q_2,$$

The definition of a state differentiator implies directly that when an automaton is reduced it is closed under congruences.

On the other hand an automaton is ground if all states labeled by the data type τ are also labeled by one of τ's constructors.

Definition 6.2.8 (Ground states and ground automata) *A state q in the automaton $\mathcal{A}$ is* ground *if for every state r, where $r \trianglelefteq q$ and $sort(r) = \tau$ then $\lambda(r) \in \Sigma$.*

The automaton $\mathcal{A}$ is ground *if all its states are ground.*

We use minimal ground reduced automata that are closed under selector/constructor redexes to extract models where all nodes in Q have different interpretations.

Lemma 6.2.9 *Let $\mathcal{A}$ be (1) minimal, (2) closed under constructor/selector redexes, (3) ground, and (4) reduced, and assume that base sorts $S_1, \ldots, S_n$ each have infinitely many elements, then there is an injective model $\mathcal{M} : \langle M, \Sigma^{\mathcal{M}} \rangle$ of $\mathcal{A}$:*

Injectivity	$q_1^{\mathcal{M}} = q_2^{\mathcal{M}} \;\rightarrow\; q_1 = q_2$	$\forall q_1, q_2 \in Q$
Constructors	$q^{\mathcal{M}} = \lambda(q)^{\mathcal{M}}(children(q)^{\mathcal{M}})$	$\forall q \in Q, \lambda(q) \in \Sigma$
Selectors	$c^{\mathcal{M}}(\overline{a}) = q^{\mathcal{M}} \;\wedge\; s_i^c(q) \in use(q) \;\rightarrow\; a_i = q^{\mathcal{M}}$	$\forall q \in Q, \overline{a} \in M$

Proof:

Not very surprising we can identify $\mathcal{M}$ with $\mathcal{A}$ setting $M = Q$. First, since each S_i is infinite-state, we can associate a distinct element with each state in Q whose sort belongs to one of the parameters. Since $\mathcal{A}$ is

1. minimal, then $\mathcal{M}$ is *extensional* (closed under congruence with respect to the constructors),
2. redex closed, then selectors are necessarily interpreted according to their definition,
3. ground, then every node of sort τ corresponds unambiguously to a constructor term in $\mathcal{M}$,
4. reduced, then $\mathcal{M}$ is closed under congruences with respect to the selectors.

While ground automata correspond naturally to injective models, we do not need a ground automaton to detect the existence of an injective model. The premises of the following lemma suffice:

Lemma 6.2.10 *Let $\mathcal{A}$ be a τ-automaton. If*

1. *$\mathcal{A}$ is closed under constructor/selector redexes,*
2. *$\mathcal{A}$ is reduced,*
3. *τ is non-flat and non-singular,*

then there is a ground reduced automaton closed under constructor/selector redexes $\mathcal{A}'$ and embedding $\imath : \mathcal{A} \hookrightarrow \mathcal{A}'$. By embedding we understand an $\mathcal{A}'$ that coincides with $\mathcal{A}$ where $\lambda(q) \in \Sigma$ in $\mathcal{A}$, but may relabel non-constructor states by constructors and add extra states.

Proof:

We construct $\mathcal{A}'$ as in the pure constructor case. Namely for each state q_i in $\mathcal{A}$ of sort τ not labeled by a constructor we allocate a fresh ground automaton $\mathcal{A}_i$ with state q_i', such that the term associated with q_i' is not isomorphic with any term in $\mathcal{A}$. We then merge q_i and q_i' to eliminate the non-constructor state. Repeated eliminations of non-constructor states in $\mathcal{A}$ produces the ground automaton $\mathcal{A}'$ and embedding $\imath : \mathcal{A} \hookrightarrow \mathcal{A}'$. This gives us the embedding, and that $\mathcal{A}'$ is ground.

We have to verify

1. $\mathcal{A}'$ is reduced. For this purpose we will extend every state differentiator (Π, T) for $\mathcal{A}$ to a state differentiator (Π', T') for $\mathcal{A}'$ by extending paths in Π that may in $\mathcal{A}$ end in a variable to paths in $\mathcal{A}'$ that witness the difference between the ends. In more detail, suppose the triple $(\pi, q_1, q_2) \in T$, and $\pi(q_1)$ ends in a node labeled by a non-constructor. Then $\pi(q_2)$ does not end in the same node. In $\mathcal{A}'$, $\pi(q_1)$ may again be labeled by a constructor, and in the worst case it may coincide with the constructor labeling $\pi(q_2)$. But by the construction of $\mathcal{A}'$ these nodes can be differentiated by extending π.
2. $\mathcal{A}'$ is closed under constructor/selector redexes. This follows as the new states in $\mathcal{A}'$ do not introduce any new redexes.

Lemma 6.2.10 now implies that if the data type is infinite domain, the elimination of selector redexes in algorithm 6.8 produces either an automaton $\mathcal{A}$ from which a model for constraints including disequalities can be extracted, or establishes the unsatisfiability of the given constraints.

6.2.5 Subterm relations

We now move to adding ground support for subterm relationships of the form $s \preceq t$, meaning s is a subterm of t. Negated constraints $s \not\preceq t$ are naturally also admitted.

The ground case for recursive constructor terms is shown NP-complete in [Ven87]. Membership in NP is established by showing that every satisfiable set of constraints has a model of cubic size. Compared to search-based decision procedures, this is highly impractical. NP-hardness carries directly over to co-recursive data types. A search-based decision method for co-recursive terms is presented in [Tul94], which shows how this decision procedure can also be modified to handle the recursive case. However, that decision procedure requires normalized terms of the following form:

$$t = v, \quad t \neq v, \quad t \preceq v, \quad t \not\preceq v \ . \tag{6.10}$$

A conjunction L of these terms is satisfiable (for the class of non-degenerate co-recursive data types) if and only if the following two tests are passed:

T1 It is not the case that $v \sqsubseteq w$ and $v \not\preceq w \ \in \ L$.

T2 It is not the case that $t \preceq v \ \in \ L$, $s \not\preceq w \ \in \ L$, $v \sqsubseteq w$, and $q_s \trianglelefteq q_t$.

where

Definition 6.2.11

- *vRw iff there is a $t \preceq w \ \in \ L$ and $v \in FV(t)$.*
- $\rightarrow \equiv \ R^+$
- $\sqsubseteq \equiv \ R^*$

Using the τ-automaton data-structure we will present a realization of the decision procedures for recursive and co-recursive data types. Our procedure transforms conjunctions of constraints of the form:

$$t = s, \quad t \neq s, \quad t \preceq s, \quad t \not\preceq s \ .$$

into a disjunction of solved form constraints $\bigvee_i L_i$, where each L_i is a conjunction of:

$$t \neq s, \quad t \preceq v, \quad t \not\preceq s \ . \tag{6.11}$$

together with a τ-automaton $\mathcal{A}$ representing all equalities. To represent such constraints we will use the tuple

$$\langle \mathcal{A}, \ \mathcal{D}, \ \mathcal{I}, \ \mathcal{N} \rangle \tag{6.12}$$

where

- $\mathcal{A}$ is a minimized automaton representing all terms and equalities in the constraint set L.

- $\mathcal{D} \subseteq Q \times Q$ corresponds to disequalities.
- $\mathcal{I} \subseteq Q \times Q$ corresponds to inequalities $t \preceq s$.
- $\mathcal{N} \subseteq Q \times Q$ corresponds to negated inequalities $t \not\preceq s$.

Notice here, that the only solved form conversion required is of $t \preceq s$ into cases $t \preceq v$. The solved form conversion consists of saturating with respect to the following steps:

1. Using the procedures for handling equalities and disequalities we can represent $\mathcal{A}$ as the minimal automaton satisfying all equalities in L, and differentiating all asserted disequalities $\mathcal{D}$. Thus, $\mathcal{A}$ is the resulting automaton after an invocation of the algorithm in Figure 6.7 applied to equalities and disequalities. If it reports failure, then the constraints L are trivially unsatisfiable.

2. We need to convert $\mathcal{I}$ into a subset of $Q \times \{q \in Q \mid \lambda(q) \notin \Sigma\}$. This can be achieved using a conversion into disjunctions using the characterization of $\preceq$:

$$s \preceq t \;\leftrightarrow\; \left(\bigvee_{\substack{q \trianglelefteq q_t, \\ \lambda(q) \in \Sigma}} q_s = q \;\vee \bigvee_{\substack{q \trianglelefteq q_t, \\ \lambda(q) \notin \Sigma}} q_s \preceq q \right)$$

   This step splits the original set of constraints L into $\bigvee_i L_i$, and constitutes the computationally expensive step (the decision problem is after all NP-complete). Notice that some of the branches impose equality constraints, that modify $\mathcal{A}$ further.

3. For each L_i we build an auxiliary graph $\mathcal{G}$, whose vertices are Q (the same states as $\mathcal{A}$), and whose edges $\mathcal{E}$ are induced by δ as well as the literals asserting inequalities (the set $\mathcal{I}$):

$$(q_1, q_2) \in \mathcal{E} \quad \leftrightarrow \quad \delta(q_2) = \langle \ldots q_1 \ldots \rangle \;\text{ or }\; (q_1, q_2) \in \mathcal{I}$$

   The tests **T1** and **T2** are now replaced by the test

   N: For each pair $(q, r) \in \mathcal{N}$ (corresponding to $q \not\preceq r$) if there is a path $\langle q, q_1, \ldots q_n, r \rangle$ in $\mathcal{G}$, then the set of constraints is unsatisfiable.

It may not be impossible to establish a correspondence between Tulipani's tests and the above saturation rules and then reuse Tulipani's results. The heavy notation in [Tul94] resulting in some confusion as to what assumptions are used to establish which properties makes a direct and straight-forward proof of completeness more desirable.

Soundness is obvious by inspecting each step.

Theorem 6.2.12 (Soundness) *L is inconsistent in the theory of recursive and co-recursive data types if all branches obtained by applying rules 1-3 are unsatisfiable.*

For the theory of co-recursive data types (rational and infinite trees) we can state:

Theorem 6.2.13 (Co-recursive Completeness I) *Suppose τ is non-linear, and contains either two non-linear constructors or has a constructor $c : \cdots \times S \times \cdots \to \tau$, where $|S|$ is infinite, then L is satisfiable if and only if some branch obtained by saturating with respect to rules 1-3 is non-contradictory.*

Proof:

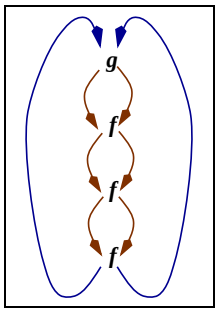
Let $X = \{x_1, \ldots, x_m\}$ be the states in $\mathcal{A}$ that are not labeled by constructors. We extend $\mathcal{A}$ to a ground automaton $\mathcal{A}'$ by assigning different ground terms to each variable in X such that every subterm relation in $\mathcal{I}$ is satisfied, and such that whenever for nodes q, r in $\mathcal{A}$, there is a path from q to r in $\mathcal{A}'$ if and only if there is a path from q to r in $\mathcal{A}$ already. This ensures that all constraints in $\mathcal{N}$ are satisfied by $\mathcal{A}'$.

Since τ is non-singular there is a binary constructor $f : \cdots \tau \times \cdots \times \tau \cdots \to \tau$. For future notational convenience we fix a binary version of f, by choosing arbitrary parameters for the domain values of f that are not of sort τ and group the arguments of sort τ in two parts. For example if f' has arity $f' : S_1 \times \tau \times \tau \times \tau \to \tau$, and $s_1 \in S_1$, we set $f(x, y) := f'(s_1, x, y, y)$.

If τ contains two non-linear constructors f' and g', let f and g be their binary versions and for each natural number n, consider the system:

$$\begin{array}{rcl} y_0^n & = & g(y_1^n, y_1^n) \\ y_1^n & = & f(y_2^n, y_2^n) \\ & \vdots & \\ y_n^n & = & f(y_0^n, y_0^n) \end{array}$$

The case $n = 3$:



Each system has a unique solution by the solution lemma, and furthermore

$$y_j^i = y_l^k \quad \leftrightarrow \quad i = k \ \wedge \ j = l \tag{6.13}$$

$$y_j^i \preceq y_l^k \quad \leftrightarrow \quad i = k \tag{6.14}$$

We therefore have a sufficient supply of different terms to allocate fresh signature terms $sig_1, \ldots sig_m$ for each variable in X, none being a subterm of each other or of any term in $\mathcal{A}$ (by choosing instances of y_0^n to represent sig_i where $n \geq |Q|$).

If τ on the other hand contains a constructor c whose domain contains an infinite S we create signatures by choosing fresh elements from S. For example, if $c : \tau \times S \times$

$\tau \to \tau$, set for each $i = 1, \ldots, m$, sig_i the unique term satisfying $sig_i = c(sig_i, s_i, sig_i)$ where the s_i are different.

We can now construct ground realizations for the variables X by building terms such that q is a subterm of $x \in X$ if and only if there is a path from q to x already in $\mathcal{A}$. Simply, let $q_1, \ldots, q_k$ be the states in $\mathcal{A}$ where $(q_1, x) \in \mathcal{I}, \ldots, (q_k, x) \in \mathcal{I}$ and set

$$\delta(x) := f(q_1, f(q_2, \ldots, f(q_k, sig_x))) \ .$$

This defines $\mathcal{A}'$.

By construction all constraints in $\mathcal{I}$ are satisfied. Also the signature terms ensure that different states in $\mathcal{A}'$ are not bisimilar, so $\mathcal{A}'$ is minimal. Finally, we have to verify that all constraints in $\mathcal{N}$ are satisfied. Let $(q, r) \in \mathcal{N}$, then by saturation step 3, $q \neq r$, and there is no path in $\mathcal{A}$ from q to r. We must show that there is no path in $\mathcal{A}'$ from q to r. But by the construction of $\mathcal{A}'$ this could only be the case if q is a subterm of one of the new states. However q is not a subterm of sig_x for any x, and since sig_x is only a subterm of q if there is a path in $\mathcal{A}$ from x to q, so q is a subterm of x in $\mathcal{A}'$ iff there is a path from x to q in $\mathcal{A}$.

▟

The assumptions of the theorem are necessary. For instance with the signature

$$\langle \tau, \mathtt{NIL} : \tau, \mathtt{CONS} : \tau \times \tau \to \tau \rangle \tag{6.15}$$

the following constraints

$$\Omega = \mathtt{CONS}(\Omega, \Omega) \ \wedge \ \mathtt{NIL} \not\leq x \ \wedge \ \Omega \not\leq x$$

are not satisfiable, but saturation fails to detect this. The remaining cases thus consists of signatures of the form

$$\langle \tau, \ \mathtt{NIL}_1 : \tau, \ldots, \mathtt{NIL}_k : \tau, \ \mathtt{CONS} : \tau \times \tau \to \tau \rangle \tag{6.16}$$

The constant atomic terms of this signature is the set

$$constants : \quad \{\mathtt{NIL}_1, \ldots, \mathtt{NIL}_k, \Omega\} \qquad \text{where } \Omega = \mathtt{CONS}(\Omega, \Omega) \ .$$

These are the terms that must be part of any minimal ground automaton.

To cover the remaining cases we maintain a set $sigAvoid(x)$ with each state in $\mathcal{A}$, which is initially empty, and add an additional saturation rule:

4. When $(q, r) \in \mathcal{N}$ and q is ground, let $x_1, \ldots, x_n$ be the variables that can reach r in $\mathcal{G}$. Set
$$sigAvoid(x_i) := sigAvoid(x_i) \ \cup \ \{q\} \qquad \text{for each } i$$
If for any x_i
$$constants \setminus sigAvoid(x_i) = \{\Omega\}$$

assert $x_i = \Omega$.

If for any x_i

$$constants \setminus sigAvoid(x_i) = \emptyset$$

report unsatisfiability.

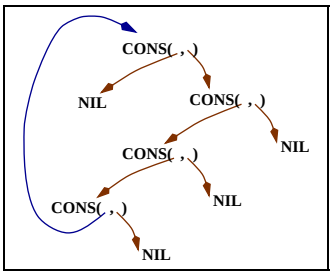
The completeness proof 6.2.13 can now be extended

Theorem 6.2.14 (Co-recursive Completeness II) *Suppose τ is non-linear and has a signature isomorphic to (6.16), then L is satisfiable if and only if some branch obtained by saturating with respect to rules 1-4 is non-contradictory.*

The proof is analogous, except this time we use the supply

$$\begin{array}{rcl} y_0^n & = & \texttt{CONS}(\texttt{NIL}_i, y_1^n) \\ y_1^n & = & \texttt{CONS}(y_2^n, \texttt{NIL}_i) \\ & \vdots & \\ y_n^n & = & \texttt{CONS}(y_0^n, \texttt{NIL}_i) \end{array}$$

The case $n = 3$:



of different incompatible terms for sig_x, when $\texttt{NIL}_i \in constants \setminus sigAvoid(x)$.

In the case of recursive data types, the subterm relationship is a partial order, and may imply additional equality constraints. Thus, strongly connected components of $\mathcal{G}$ are collapsed by asserting equalities between the nodes in each component. As discussed in 6.2.3.1 the unification algorithm for recursive data types provides a built-in occurs check which reports unsatisfiability in the presence of a cycle traversing a constructor.

In complete analogy with co-recursive completeness we have:

Theorem 6.2.15 (Recursive Completeness I) *Suppose τ is non-linear and contains a non-recursive constructor $c : S_{i_1} \times \cdots \times S_{i_k} \to \tau$ where one of the domain sorts is infinite, then L is satisfiable if and only if some branch obtained by saturating with respect to rules 1-3 is non-contradictory.*

When it is only possible to supply a finite set of non-recursive constructor terms the set *constants* is finite and we can under certain circumstances use step 4 (without the condition involving Ω) and easily state:

Theorem 6.2.16 (Recursive Completeness II) *Suppose τ is non-linear and contains a constructor $c : \cdots \times S \times \cdots \times \tau \times \cdots \to \tau$, where $|S|$ is infinite, then L is satisfiable if and only if some branch obtained by saturating with respect to rules 1-4 is non-contradictory.*

This follows as we can here build unique sig_x using the non-recursive constructors that are not in $sigAvoid(x)$ to form the leaves of sig_x and by using unique versions of c to distinguish the signatures.

The most complicated case is when there is essentially only a finite number of constructors (both recursive and non-recursive). For instance, the constraints below of the signature (6.15) are unsatisfiable

$$\texttt{CONS}(\texttt{NIL},\texttt{NIL}) \not\preceq x \ \wedge \ \texttt{NIL} \neq x$$

because every finite tree contains a `CONS(NIL,NIL)` leaf. In general we need an essentially stronger version of saturation step 4 to handle these cases:

4' When $(q,r) \in \mathcal{N}$ and q is ground, let $x_1, \ldots, x_n$ be the variables that can reach r in $\mathcal{G}$. Set

$$\mathit{sigAvoid}(x_i) := \mathit{sigAvoid}(x_i) \ \cup \ \{q\} \qquad \text{for each } i$$

Suppose

1. $\mathit{sigAvoid}(x_i) \neq \emptyset$,
2. $q \in \mathit{sigAvoid}(x_i)$ is a term of maximal size in that set,
3. for all terms of length $|q|$ there is some $r \in \mathit{sigAvoid}(x_i)$ such that $r \preceq q$.

split L into branches, one for each term t of length less than $|q|$ that is not a subterm of any term in $\mathit{sigAvoid}(x_i)$.

The extra case splitting ensures that every remaining variable x_i in a non-contradictory branch admits arbitrary large signatures by choosing a term t_i not in $\mathit{sigAvoid}(x_i)$ of maximal length and extending it using the non-linear constructor c as much as desired. For instance, $c(t_i, c(t_i, \ldots, c(t_i, t_i)))$. To ensure that none of the signatures are subterms of each other let n be the size of $\mathcal{A}$ ($|Q|$) and create the signatures

$$\begin{aligned}
\mathit{sig}_1 : &\quad c(c(t_1,t_1), \underbrace{c(t_1, c(t_1, \ldots, c(t_1,t_1)))}_{n+1}), \\
\mathit{sig}_2 : &\quad c(c(t_2,t_2), \underbrace{c(t_2, c(t_2, \ldots, c(t_2,t_2)))}_{n+2}), \\
&\qquad\qquad \vdots \\
\mathit{sig}_i : &\quad c(c(t_i,t_i), \underbrace{c(t_i, c(t_i, \ldots, c(t_i,t_i)))}_{n+i})
\end{aligned} \tag{6.17}$$

This leads us to the final result:

Theorem 6.2.17 (Recursive Completeness III) *If τ is non-linear then L is satisfiable if and only if some branch obtained by saturating with respect to rules 1,2,3, and 4' is non-contradictory.*

6.2.5.1 The first-order theory of subterms

The first-order theory of equality and subterm relations cannot be easily encoded into wS2S. In fact it is undecidable [Ven87, Tre92] when there is at least one ternary constructor.

Undecidability is established by reducing arbitrary instances of the Post correspondence problem to a statement in the first-order theory of recursive data types with the subterm relation. See also [Com90], which provides ground decision procedures of lexicographic orderings. The first-order extension is later proved undecidable by the same author. A conjecture raised in [Ven87] is that the first-order theory of finite binary trees with subterm relation ($\langle \mathtt{NIL}, \mathtt{CONS}, =, \preceq \rangle$) is decidable. It is somewhat surprising that neither [Ven87] nor [Tre92] realize that the Post correspondence problems can also be reduced to the first-order theory of finite binary trees with subterms. The construction given below is different from the case when the signature contains a ternary constructor, so we give it here in all details to settle the conjecture (in the negative).

Theorem 6.2.18 *The first-order theory of finite binary trees with subterm relation is undecidable.*

Proof:

Take an instance of the Post correspondence problem (a Post system), which consists of a finite set of pairs of strings $(v_1, w_1), \ldots, (v_n, w_n)$ over the alphabet $\{0, 1\}$ and asks if there is a sequence $i_1, \ldots, i_k$ of indices ranging over $\{1, \ldots, n\}$, such that $v_{i_1} v_{i_2} \cdots v_{i_k} = w_{i_1} w_{i_2} \cdots w_{i_k}$. There is no effective procedure which takes as input an arbitrary Post system and provides an answer whether there exists such a sequence or not. For each Post system we now construct a formula over the theory of finite binary trees with subterm relations which is valid if and only if there is a solution to the given system. For this purpose we will give an encoding procedure which can record the set of string pairs (v, w) that are obtained from a finite set of indices $i_1, \ldots, i_k$ such that $v = v_{i_1} v_{i_2} \cdots v_{i_k}$ and $w = w_{i_1} w_{i_2} \cdots w_{i_k}$. The given Post system is then solvable if there is a tree with a pair (v, w) where $v = w \neq \epsilon$.

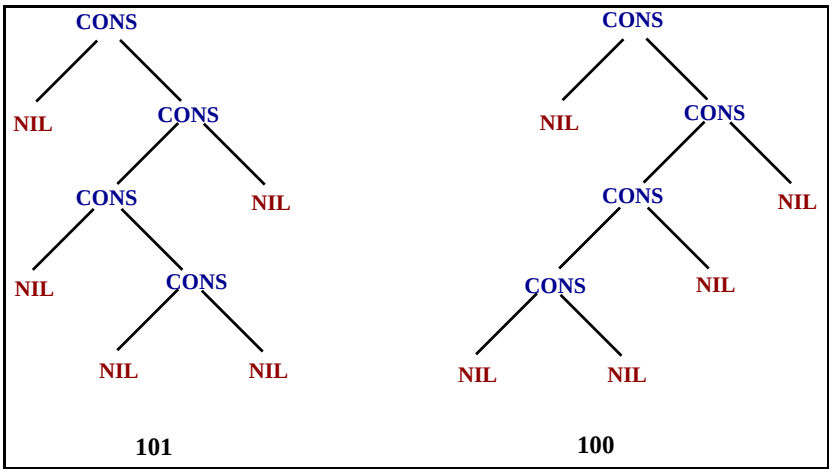
Define $\mathit{string}(u)$ if u is not $\mathtt{NIL}$ and every branching point in u has at least one branch being $\mathtt{NIL}$.

$$\mathit{string}(u) \stackrel{\text{def}}{=} u \neq \mathtt{NIL} \wedge \forall x \preceq u \,.\, x = \mathtt{NIL} \vee \exists y \,.\, x = \mathtt{CONS}(y, \mathtt{NIL}) \vee x = \mathtt{CONS}(\mathtt{NIL}, y)$$

The binary trees that are strings will be used to encode strings over the alphabet $\{0, 1\}$, and the empty string is encoded via $\mathtt{CONS}(\mathtt{NIL}, \mathtt{NIL})$. If u is any term and v is a sequence of 0's and 1's we define the concatenation $u \cdot v$ by:

$$\begin{array}{lcl} u \cdot \epsilon & \stackrel{\text{def}}{=} & u \\ u \cdot 0v & \stackrel{\text{def}}{=} & \mathtt{CONS}(u, \mathtt{NIL}) \cdot v \\ u \cdot 1v & \stackrel{\text{def}}{=} & \mathtt{CONS}(\mathtt{NIL}, u) \cdot v \end{array}$$

Clearly if u satisfies $\mathit{string}(u)$ then also $\mathit{string}(u \cdot v)$. We can finally convert an entire string u into a binary tree representing it by defining $\epsilon \stackrel{\text{def}}{=} \mathtt{CONS}(\mathtt{NIL}, \mathtt{NIL})$ and compute $\epsilon \cdot u$. Sample encodings of strings 101 and 100 are illustrated below.



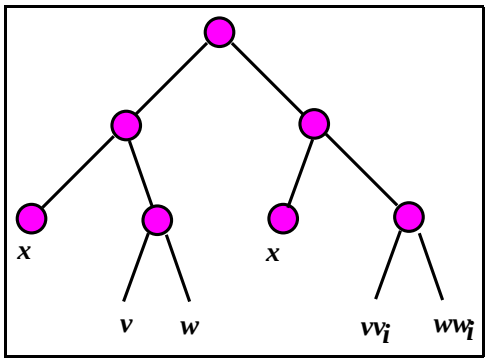
Trees representing a pair (v, w) and sequence $i_1, \ldots, i_k$ with $v = v_{i_1} v_{i_2} \cdots v_{i_k}$ and $w = w_{i_1} w_{i_2} \cdots w_{i_k}$ are captured by the predicate *Root*. To make the definition of *Root* less painful to read we will also introduce two auxiliary predicates *LHS*, *RHS* for the immediate left and right branches of terms satisfying *Root*. Informally $Root(u)$ holds if and only if u records a pair (v, w) and corresponding history of indices used to form v and w.

$$Root(u) \stackrel{\text{def}}{=} \exists \begin{array}{l} x, \\ v, \\ w, \\ v', \\ w' \end{array} . \quad \begin{array}{ll} & u = \texttt{CONS} \left(\begin{array}{l} \texttt{CONS}(x, \texttt{CONS}(v, w)), \\ \texttt{CONS}(x, \texttt{CONS}(v', w')) \end{array} \right) \\ \wedge & LHS(x, v, w) \\ \wedge & RHS(v, w, v', w') \end{array} \tag{6.18}$$

$$LHS(x, v, w) \stackrel{\text{def}}{=} \begin{array}{ll} & v = w = \epsilon \ \wedge \ x = \texttt{NIL} \\ \vee & \exists x', y' \ . \quad x = \texttt{CONS} \left(\begin{array}{l} \texttt{CONS}(x', y'), \\ \texttt{CONS}(x', \texttt{CONS}(v, w)) \end{array} \right) \\ & \wedge \ string(v) \ \wedge \ string(w) \end{array} \tag{6.19}$$

$$RHS(v, w, v', w') \stackrel{\text{def}}{=} \bigvee_{i=1}^{n} v \cdot v_i = v' \ \wedge \ w \cdot w_i = w' \tag{6.20}$$

The branching on a *Root* is illustrated below



We can require recursively that the repeated branch x in the definition of *Root* by requiring that every non-NIL subterm of u satisfying *LHS* for some v, w, is again a *Root*. This form of implicit recursion is much similar to the S2S encoding of recursive and co-recursive data types. Hence, define

$$GoodRoot(u) \stackrel{\text{def}}{=} \overset{Root(u)}{\wedge} \left(\forall x \preceq u \;.\; \left(\exists \begin{smallmatrix} v, \\ w \end{smallmatrix} \;.\; LHS(x,v,w) \;\wedge\; x \neq \texttt{NIL} \right) \right) \\ \rightarrow \; Root(x) \qquad (6.21)$$

Finally, the given Post system is solvable if and only if

$$\exists u \;.\; GoodRoot(u) \;\wedge\; \exists x, y, v \;.\; v \neq \epsilon \;\wedge\; u = \texttt{CONS}(x, \texttt{CONS}(y, \texttt{CONS}(v, v)))$$

One direction is trivial, given a solution to a Post system we construct a *GoodRoot* encoding the sequence of productions that led to the solution. In the other direction well-founded induction on terms satisfying *GoodRoot* establishes that they represent only legal applications of the Post production rules and whenever $u = \texttt{CONS}(x, \texttt{CONS}(y, \texttt{CONS}(v, w)))$ for some x, y, v, w, then $string(v)$ and $string(w)$. ∎

6.2.6 Taking lengths of recursive data types

While the subterm relation is a natural special relation to support for (co-)recursive data types, a *length* accessor seems to be a reasonable utility to add to recursive data types. It is especially relevant in termination arguments for recursive programs. The length of term t, written $|t|$, is interpreted as the number of constructors used to form the term t. Thus, we have the corresponding axiomatization in (6.9).

The thrill in adding this seemingly innocent utility is that constraints with equality, disequality, and subterm relationships on data types can now depend directly on the theory of integer (linear) arithmetic and recursive data types. While each ground theory is decidable, checking satisfiability in isolation no longer suffices. For example given the hybrid

$$\forall c \in \Sigma, \forall (y_1, \ldots, y_n) \in \mathbf{dom}(c) \ . \ |c(y_1, \ldots, y_n)| = 1 + \sum_{i=1,\ sort(y_i)=\tau}^{n} |y_i| \qquad \text{(length)}$$

Figure 6.9: Length accessor axiomatization

constraint

$$|a| < |b| \ \wedge \ b \preceq a, \tag{6.22}$$

where a and b are variables over a recursive data type, the first constraint $|a| < |b|$ is most naturally maintained by an integer linear arithmetic solver, whereas $b \preceq a$ is maintained by the solvers presented in Section 6.2.5. Each constraint in isolation is satisfiable, but the combined constraint is clearly not. In an initial attempt we can saturate constraints via the special relation rule

$$x \preceq y \ \rightarrow \ |x| \leq |y| \ . \tag{6.23}$$

Saturating with this special relationship on (6.22) we obtain a contradiction as $|b| \leq |a|$ is added. But this does not provide in itself a complete integration.

Example: Consider the data type of trees, from example 6.1.1, where S is a singleton sort with only element $\bullet$, and the constraints

$$\begin{array}{ll} & |x| = 5 \\ \wedge & x \neq node(node(leaf(\bullet), leaf(\bullet)), leaf(\bullet)) \\ \wedge & x \neq node(leaf(\bullet), node(leaf(\bullet), leaf(\bullet))) \ . \end{array}$$

These are unsatisfiable as the only terms of length 5 are the ones x is required to be different from.

Example: Regardless of the choice of recursive data type the constraints

$$t \preceq x \ \wedge \ s \preceq x \ \wedge \ |x| < |t| + |s| \ \wedge \ t \not\preceq s \ \wedge \ s \not\preceq t$$

are unsatisfiable.

Example:

$$\langle \tau, \mathtt{NIL} : \tau, f : \tau \times \tau \times \tau \times \tau \rightarrow \tau \rangle$$

All terms of τ have length $1 + 4x$ for some x.

6.2.6.1 Decidability

The first question is naturally whether the combined theory is decidable. We claim it is, though the outline we provide does not encourage a search oriented implementation.

Theorem 6.2.19 *Validity in the universal theory of integer linear arithmetic and recursive data types with lengths is decidable.*

As usual, validity of a formula is established by checking unsatisfiability of a negation of that formula. To check satisfiability of a set of data type and arithmetical constraints $\mathcal{C}$ we will perform a collection of saturation rules, that split $\mathcal{C}$ into a finite number of disjunctions. Each disjunction is normalized to the form $L \wedge LA$, where L is a set of data type constraints saturated with respect to the procedure from 6.2.5, and L is satisfiable in some model for the theory of data types; LA is a set of linear arithmetic inequalities, is satisfiable in a model for the theory of integer (linear) arithmetic. We then establish that if we reach a disjunct $L \wedge LA$ that cannot be split any further, then the combined set of constraints for that disjunct is satisfiable.

We shall consider the case where τ is non-linear and has a finite number of constructors (cf. Theorem 6.2.17). To simplify the discussion, but without losing generality, we shall consider signatures for τ with two non-linear constructors and an arbitrary number of non-recursive constructors. This summarizes the general case. Suppose therefore that the signature of τ is of the form

$$\langle \tau,\ \mathtt{NIL}_1 : \tau, \ldots, \mathtt{NIL}_k : \tau,\ f : \underbrace{\tau \times \cdots \times \tau}_{a} \to \tau,\ h : \underbrace{\tau \times \cdots \times \tau}_{b} \to \tau,\ \rangle$$

- Euclid's algorithm for computing greatest common divisors provides m and n such that
$$\gcd(a,b) = ma - nb$$
Set $g = \gcd(a,b)$, then we can create terms of length $1+\frac{a}{g}nb$, $1+\frac{a}{g}nb+g$, $1+\frac{a}{g}nb+2g, \ldots$ using constructors f and h, because we can write the numbers $\frac{a}{g}nb, \frac{a}{g}nb+g, \ldots$ as linear combinations of a and b:
$$\begin{array}{lcll} \frac{a}{g}nb, & & & \\ \frac{a}{g}nb+g & = & ma+(\frac{a}{g}-1)nb, & \\ \frac{a}{g}nb+2g & = & 2ma+(\frac{a}{g}-2)nb, & \\ \ldots, & & & \\ \frac{a}{g}nb+\frac{a}{g}g & = & \frac{a}{g}ma & = \ \frac{a}{g}nb+a \end{array}$$
and each linear combination of a and b corresponds to a term using f and h by applying f and h as many times as the coefficients of a and b dictate.
For future reference abbreviate
$$k \stackrel{\text{def}}{=} \frac{a}{g}nb\ .$$

- Let $x_1, \ldots, x_n$ be a permutation of the variables in L, and guess an ordering:

$$0 < |x_1| = |x_2| < |x_3| < \ldots = |x_n|$$

  The amount of guessing is finite as there are $n!$ permutations of the x_i and 2^{n-1} ways to choose between equality or strict inequality. Add the ordering constraints to LA.

- Saturate L with respect to rules 1,2,3,4'.

- For $|x_i| = |x_{i+1}|$ split on the constraints

  1. $x_i = x_{i+1}$. This eliminates one variable.
  2. $x_i \not\preceq x_{i+1},\ x_{i+1} \not\preceq x_i$. This ensures that they cannot be subterms of each other.

- Saturate with respect to the special relations rule (6.23).

- We will now eliminate the variables in the same order as the size ordering constraints that were guessed above.

$$\begin{array}{lcl} |sig_1| + \sum\limits_{(q,x_1)\in\mathcal{I}} (|q| + a - 1) + k & = & l_1 \\ & \vdots & \\ |sig_i| + \sum\limits_{(q,x_i)\in\mathcal{I}} (|q| + a - 1) + k & = & l_i + \sum\limits_{j=1}^{i-1} a_i^j |x_j| \\ & \vdots & \end{array}$$

  1. The factor $|sig_i|$ provides the space required to build a unique signature as defined in (6.17).
  2. The factor $\sum\limits_{(q,x_i)\in\mathcal{I}} (|q| + a - 1)$ provides the space required to satisfy the subterm relations on top of the signature.
  3. The factor k provides the space to build on top of 2 to generate terms within distance g.

  Now split into the cases:

$$\begin{array}{ll} \text{case 1}: & |x_1| < l_1 \\ \text{case 2}: & |x_1| \geq l_1 \ \wedge\ |x_1| = 1 + y^1_{new} g \end{array}$$

  In the first case there are finitely many ways to construct x_1, so x_1 can be eliminated from these branches producing constraints with fewer variables.

In general we split on the cases

$$\text{case 1}: \quad |x_i| < l_i + \sum_{j=1}^{i-1} a_i^j |x_j|$$
$$\text{case 2}: \quad |x_i| \geq l_1 + \sum_{j=1}^{i-1} a_i^j |x_j| \ \wedge \ |x_i| = 1 + y_{new}^i g$$

In every instance i the first branch eliminates a variable by enumerating all possible ways to form x_i.

Suppose therefore that all branches choose the second case and that the integer constraints are satisfiable. Then, the constraints are satisfiable in a model where all lower bounds are respected. But, the lower bounds were chosen such that different x_i could be realized of any size exceeding the lower bound and such that none of the x_i's were subterms of each other unless explicitly required by the constraints in L.

6.2.6.2 An incomplete search-oriented procedure

Our implementation uses an incomplete procedure based on the saturation rules:

subterm saturation Whenever $s \preceq v, t \preceq v$ are in L, then either $t \preceq s$ or $t \not\preceq s$ are in or implied by L.

lower bounds Whenever $|v| \in \mathbf{dom}(L_{\mathcal{T}})$, then $0 \leq |v|$ is implied by LA.

special relation: forward Whenever $t \preceq v$ is in L and $|v| \in \mathbf{dom}(L_{\mathcal{T}})$, then $\sigma_\tau(|t|) \leq |v|$ is implied by LA.

special relation: backward Assuming subterm and lower bound saturation, if $t_1 \preceq v$, $\ldots$, $t_n \preceq v$ are the (different and incompatible) lower bounds on v in L, $n \geq 0$ ($n = 0$ is allowed) and

$$i \ = \ \texttt{INF}(LA, |v| - \sum_{i=1}^{n} |t_i|) \qquad s \ = \ \texttt{SUP}(LA, |v| - \sum_{i=1}^{n} |t_i|) \ .$$

If s is finite, i.e.,

$$0 \ \leq i \ \leq \ s \ < \ \infty$$

then for each term skeleton $t(x_1, \ldots, x_n)$ with unique occurrences of the free variables $x_1, \ldots, x_n$, such that $|t(x_1, \ldots, x_n)| = k + |x_1| + \ldots + |x_n|$ for some $i \leq k \leq s$ replace v by $t(t_1, \ldots, t_n)$ to obtain a set of new constraints without v. Thus, set the updated constraints to $(L \wedge LA)[v \mapsto t(t_1, \ldots, t_n)]$ for each of the possible ts.

If the set of terms satisfying $|t(x_1, \ldots, x_n)| = k + |x_1| + \ldots + |x_n|$ for some $i \leq k \leq s$ is empty, then the set of constraints is unsatisfiable. This could for instance be the case when $s < 0$.

The saturation rules are not complete as the example with the 4-ary constructor witnesses. The single constraint $|x| = 2y$ is simply not satisfiable.

6.3 Open problems

1. Is the full first-order theory of infinite trees decidable when all constructors have arity no larger than 1? The problem is posed in [Tul94].

6.4 Summary

This chapter presented decision procedures for data-types, including the sub-term relation and length constraints. Both recursive as well as co-recursive data-types have been examined and we have shown how these theories can be integrated within the solver-based combination of decision procedures.

Chapter 7

Bit-vectors

Bit-vectors are the natural data-type for hardware descriptions. To handle bit-vectors in computer-aided verification, it is convenient to have specialized decision procedures to solve constraints involving bit-vectors and their operations.

To verify hardware designs, Mark Pichora developed a compiler from the Verilog hardware description language to fair transition systems. Since bit-vectors are pervasive in Verilog we have found it useful to develop the decision procedures for bit-vectors described in this chapter. The presented procedure is easy to integrate tightly within the combination decision procedures for other theories, which fits well into the wide scope of STeP.

An algorithm that addresses bit-vectors from a perspective similar to ours has been reported in [CMR97]. In an effort to use that algorithm we were unable to reconstruct the rutines necessary to handle bit-wise boolean operations. We devised an algorithm where bit-wise boolean operations could be easily handled. The key feature of the procedure is that it often only splits contiguous bit-vectors on demand. Its performance is often independent of the length of the bit-vectors in the input. We also briefly discuss non-equational bit-vector constraints, which had not received proper attention elsewhere.

Legal inputs to the STeP-Verilog verification tool include *parameterized* hardware designs where the bit-vector size is not fixed at verification time. The potential need then arises for a method that can handle both fixed and non-fixed size bit-vectors. In certain cases our simple procedure for fixed size bit-vectors can be used directly for non-fixed size bit-vectors. To handle more cases, we first present an optimized decision procedure for equations $s = t$, where s and t do not contain bit-wise boolean operations, and then extend it to handle bit-vectors whose sizes are parameterized (still without supporting boolean operations). To our knowledge this was the first reported decision procedure that handles concatenation of a non-trivial class of non-fixed size bit-vectors. Independent of this effort, however, Möller and Rueß [MR98] developed much similar transformation rules applying to a larger set of equality constraints, but without being able to give a termination argument (albeit, it is a non-trival problem). With a different starting point [BDL98] give optimized procedures for handling bit-vector arithmetic.

7.1 Bit-vectors

Bit-vector terms are of the form

$$\begin{array}{lll} t & ::= & x \mid t[i:j] \mid t_1 \circ t_2 \mid c_{[m]} \mid t_1 \; op \; t_2 \\ op & ::= & \& \text{ (bitwise and)} \mid \; \hat{} \; \text{ (bitwise xor)} \mid \text{“|” (bitwise or)} \\ c & ::= & 1 \mid 0 \end{array}$$

$t[i:j]$ denotes subfield extraction, and $\circ$ concatenates two bit-vectors. The constant 0 is synonymous with false and 1 with true. For clarity, a term may be annotated by a length, such that $t_{[m]}$ indicates that t has length m.

Terms are well-formed when for every subterm $t_{[m]}[i:j]$, $0 \leq i \leq j < m$, and for every $s_{[m]} \; op \; t_{[n]}$, $n = m$. Terms without occurrences of op are called *basic bit-vector terms.*

Bit-vectors can be interpreted as finite functions from an initial segment of the natural numbers to booleans. Hence, if η is a mapping from bit-vector variables $x_{[m]}$ to an element of the function space $\{0, \ldots, m-1\} \to \mathcal{B}$ we interpret composite terms as follows:

$$\begin{array}{lcl} [\![x]\!]_\eta & = & \eta(x) \\ [\![t[i:j]]\!]_\eta & = & \lambda k \in \{0, \ldots, j-i\}.[\![t]\!]_\eta(i+k) \\ [\![s_{[m]} \circ t_{[n]}]\!]_\eta & = & \lambda k \in \{0, \ldots, m{+}n{-}1\}.\textbf{if} \;\; k < m \;\; \textbf{then} \;\; [\![s]\!]_\eta(k) \;\; \textbf{else} \;\; [\![t]\!]_\eta(k{-}m) \\ [\![s_{[m]} \; op \; t]\!]_\eta & = & \lambda k \in \{0, \ldots, m-1\}.[\![s]\!]_\eta(k) \; [\![op]\!] \; [\![t]\!]_\eta(k) \\ [\![c_{[m]}]\!]_\eta & = & \lambda k \in \{0, \ldots, m-1\}.c = 1 \end{array}$$

Bit-vector terms from the above grammar appear, for instance, throughout the system description and verification conditions from a split-transaction bus design from SUN Microsystems [Kam96]. A sample proof obligation encountered during STeP's verification of a safety property of the bus (namely, processes are granted exclusive and non-interfering access to the bus) takes the form

$$\begin{array}{rl} l\_wires = 4 \;\wedge\; request \neq \; 0_{[8]} \;\rightarrow & \\ \quad (request\_h \;\hat{}\; request) \neq 0_{[8]} & \\ \vee \;\; (request \neq 0_{[8]}) \wedge request = request\_h & \end{array} \tag{7.1}$$

where *request* and *request_h* are bit-vector variables of length 8. While this proof obligation is evidently valid, a simple encoding of bit-vectors as tuples causes examination of multiple branches when establishing the verification condition. The procedure developed here avoids this encoding and its potential case splitting. This and similar verification conditions can then be established independently of the bit-vector length (and in a fraction of a second). Thus, our procedure is able to establish this verification condition when the length 8 is replaced by an arbitrary parameter N.

While other logical operations like shifting can easily be encoded in the language of bit-vectors we analyze, the arithmetical (signed, unsigned and IEEE-compliant floating point) operations are not treated at all here.

7.2 Alternative approaches

As usual, a direct axiomatization can be used to establish all verification conditions we consider. Better than a raw axiomatization, proof assistants like ACL2 and PVS provide sophisticated libraries containing relevant bit-vector lemmas. But, although highly useful, libraries do not provide a decision method.

In the symbolic model checking community, BDDs [Bry86] (binary decision diagrams) are used to efficiently represent and reason about bit-vectors. Purely BDD based representation of bit-vectors requires allocating one variable for every position in a bit-vector. (Just two bit-vector variables each of length 64 require allocation of 128 variables, pushing the limits of current BDD technology). A BMD-based (binary moment diagram) representation [BC95] optimizes on this while being able to efficiently perform arithmetical operations on bit-vectors. Unfortunately it is nontrivial to combine BMDs efficiently into the Shostak combination.

Since the values of bit-vectors range over strings of 0's and 1's it is possible to use regular automata to constrain the possible values of bit-vectors. Using this approach the MONA tool [BK95] can effectively represent addition of parameterized bit-vectors using M2L (Monadic Second-Order Logic). The expressive power of M2L also allows a direct and practical decision procedure of fixed size bit-vectors encoded either as tuples of boolean variables or as unary predicates with a constant domain. Furthermore M2L allows quantification over bit-vectors (with non-elementary complexity as the price). The approach based on regular automata, however does not admit an encoding of concatenations of parameterized bit-vectors. For suppose the regular language R_x (say 10^*1) encodes evaluations of bit-vector x that satisfy constraint $\varphi(x)$. Then the pumping lemma tells us that the evaluations of y consistent with $\varphi(x) \wedge y = x \circ x$ is not in general (certainly $\{ww \mid w \in 10^*1\}$ is not) a regular language. Automata with constraints [CDG+98] (see chapter 4) is a possible remedy, but this imposes even more challenges in obtaining a direct ground integration with other decision procedures, which we seek here. Our procedure addresses this problem and solves satisfiability of ground equalities.

7.3 A decision procedure for fixed size bit-vectors

We present a normalization function $\mathcal{T}$, which takes a bit-vector term $t_{[m]}$ and a subrange (initially $[0 : m-1]$) and normalizes it to a bit-vector term $F_1 \circ F_2 \circ \ldots \circ F_n$ where each F_i is of the form

$$F ::= F \; op \; F \mid x \mid c_{[m]} \;\; .$$

A normalization routine with a similar scope can be found in [BEFS89]. In words, $\mathcal{T}$ produces a term without occurrences of subfield extraction where every $\circ$ is above every *op*. The translation furthermore maps every original variable $x_{[m]}$ to a concatenation $x_1 \circ x_2 \circ \ldots \circ x_n$, and maintains a decoding of the auxiliary variables into subranges $\mathit{decode}([i_k : j_k])$, such that $i_1 = 0$, $j_n = m-1$, and $j_k + 1 = i_{k+1}$ for $k = 1 \ldots n-1$.

The normalization function shown in Figure 7.3 is designed to satisfy the basic correspondence

$$[\![t_{[n]}]\!]\eta = [\![\mathcal{T}(t,[0:n-1])]\!]\eta'$$

for every η, where η' coincides with η on the free variables in t and, furthermore, if $\mathcal{T}$ rewrites x to $x_1 \circ \ldots \circ x_k \circ \ldots \circ x_n$, with $\mathit{decode}(x_k) = [i:j]$, then $\eta'(x_k) = \lambda k \in \{0,\ldots,j-i\}.\eta(x)(k+i)$.

Normalization works by recursive descent on the syntax tree of t, pushing a subfield extraction $[i:j]$ downwards. By maintaining only one copy of each variable, the procedure may update a variable occurrence x to a concatenation $x_1 \circ x_2 \circ x_3$ globally in the cases where only the subfield $[3:5]$ needs to be extracted from $x_{[8]}$. The result of normalizing $x[3:5]$ then becomes x_2, such that $\mathit{decode}(x_2) = [3:5]$. Since the variable x may occur in a different subterm under the scope of a boolean operator $x \;\&\; y$ the cutting of x rewrites this to $(x_1 \circ x_2 \circ x_3) \;\&\; y$. The auxiliary procedure *cut* (that takes a term and a cut-point as argument) shown in Figure 7.1 recursively cuts y in the same proportions as x, and forms the normalized concatenation $x_1 \;\&\; y_1 \;\circ\; x_2 \;\&\; y_2 \;\circ\; x_3 \;\&\; y_3$. It uses a set *parents* associated with each variable x to collect the maximal boolean subterms involving x that have already been normalized. Initially $\mathit{parents}(x) = \emptyset$ for each variable. Subterms can also be marked. By default (and initially) they are unmarked. To avoid cluttering the pseudocode we have suppressed variable dereferencing. To normalize boolean operators, $\mathcal{T}$ uses the auxiliary procedure *slice* shown in Figure 7.2, which aligns the normalized terms s and t into concatenations of equal length boolean subterms. Operator application can then be distributed over each of the equally sized portions. The auxiliary symbol ϵ is used for the empty concatenation.

The proper functioning of $\mathcal{T}$ relies on the precondition that every time $\mathcal{T}(t_{[n]},[i:j])$ is invoked, then $0 \le i \le j < n$. This ensures that whenever $\mathit{cut}(t_{[n]},m)$ is invoked then $m < n$.

Example: As an example of the translation of an bit-vector expression, consider:

$$\begin{array}{lllll} s: & w_{[7]} & \& & (y_{[7]}[0:3] & \circ\; x_{[3]}) \\ t: & y_{[7]} & \mid & (x_{[3]} \circ 1_{[1]} & \circ\; w_{[7]}[0:2]) \end{array}$$

We first apply $\mathcal{T}(s,[0:6])$ which results in cutting y into $y_1 \circ y_2$, where $\mathit{decode}(y_1) = [0:3]$, $\mathit{decode}(y_2) = [4:6]$. w is cut similarly. The translation of t results in further cutting y_1 into $y_3 \circ y_4$, where $\mathit{decode}(y_3) = [0:2]$, in order to align with $x_{[3]} \circ 1_{[1]}$. The variable $w_{[7]}$ is also cut into $w_1 \circ w_2 \circ w_3$ overing the same intervals as the parts of y, namely $[0:2],[3:3],[4:6]$. The result of translation is then:

$$s: \; w_1 \;\&\; y_3 \;\circ\; w_2 \;\&\; y_4 \;\circ\; w_3 \;\&\; x \qquad\qquad t: \; y_3 \mid x \;\circ\; y_4 \mid 1_{[1]} \;\circ\; y_2 \mid w_1$$

7.3.1 Interfacing to the Shostak combination

To canonize a term $t_{[m]}$ we first obtain $F^1 \circ \ldots \circ F^n = \mathcal{T}(t,[0:m-1])$. We will identify a free variable x_k in F^i with $x[i:j]$, where $\mathit{decode}(x_k) = [i:j]$. Each F^i is represented

```
1.  cut(F, m) =
2.      mark(F);
3.      let
4.          (F_1(x̄^1), F_2(x̄^2)) = dice(F, m)
5.      in
6.          for each j = 1, 2, x^j ∈ x̄^j do
7.              parents(x^j) := parents(x^j) ∪{F_j}
8.          return (F_1(x̄^1), F_2(x̄^2))
```

```
1.  dice(s op t, m) =
2.      let
3.          (s_1, s_2) = dice(s, m);
4.          (t_1, t_2) = dice(t, m);
5.      in
6.          return (s_1 op t_1, s_2 op t_2)
7.  dice(c_[l], m) = return (c_[m], c_[l−m])
8.  dice(x^1_[m] ∘ x^2_[n], m) = return (x^1, x^2)
9.  dice(x, m) =
10.     let
11.         [i : j] = decode(x)
12.         x_1, x_2 be fresh variables with ∅ parents
13.     in
14.         decode(x_1) := [i : i + m − 1];
15.         decode(x_2) := [i + m : j];
16.         x := x_1 ∘ x_2
17.         for each unmarked s ∈ parents(x) do
18.             s := s_1 ∘ s_2 where (s_1, s_2) = cut(s, m)
19.         return (x_1, x_2)
```

Figure 7.1: Basic cutting and dicing

```
1.  apply(op, F(x̄), G(ȳ)) =
2.      for each x ∈ x̄ ∪ ȳ do
3.          parents(x) := parents(x) \ {F, G} ∪ {F(x̄) op G(ȳ)}
4.      return F(x̄) op G(ȳ)

1.  slice(op, ϵ, ϵ) = ϵ
2.  slice(op, F(x̄)_[n] ∘ s, G(ȳ)_[m] ∘ t) =
3.          if m = n then
4.              apply(op, F(x̄), G(ȳ)) ∘ slice(op, s, t)
5.          else if m > n then
6.              (G1(ȳ1), G2(ȳ2)) := cut(G(ȳ), n);
7.              apply(op, F(x̄), G1(ȳ1)) ∘ slice(op, s, G2(ȳ2) ∘ t)
8.          else
9.              (F1(x̄1), F2(x̄2)) := cut(F(ȳ), m);
10.             apply(op, F1(x̄), G(ȳ)) ∘ slice(op, F2(x̄2) ∘ s, t)
```

Figure 7.2: Slicing and operator application

$\mathcal{T}(s\ op\ t, [i:j]) =$	$slice(op, \mathcal{T}(s,[i:j]), \mathcal{T}(t,[i:j]))$
$\mathcal{T}(s[k:l], [i:j]) =$	$\mathcal{T}(s, [k+i : k+j])$
$\mathcal{T}(s_{[n]} \circ t_{[m]}, [i:j]) =$	**if** $n \le i$ **then** $\mathrm{T}(t, [i-n : j-n])$ **else**
	if $n > j$ **then** $\mathrm{T}(s, [i:j])$ **else** $\mathcal{T}(s, [i : n-1]) \circ \mathcal{T}(t, [0 : j-n])$
$\mathrm{T}(x_{[m]}, [i:j]) =$	**if** $0 < i$ **then** $\mathcal{T}(\mathrm{second}(dice(x,i)), [0 : j-i])$ **else**
	if $j < m-1$ $(i = 0)$ **then** $\mathrm{first}(dice(x, j+1))$ **else** x
$\mathcal{T}(c_{[m]}, [i:j]) =$	$c_{[j-i+1]}$

Figure 7.3: Normalization procedure $\mathcal{T}$

in a canonical form (for instance an ordered BDD) based on a total order of the variables. A consecutive pair F^i and F^{i+1} can now be combined whenever F^i is equivalent to the boolean expression obtained from F^{i+1} by replacing each variable $x[k:l]$ by $x[k-n:l-1]$, where n is the length of F^i.

To decide the satisfiability of an equality $s_{[n]} = t_{[n]}$ and extract a canonized substitution θ we notice that $s = t$ is equivalent to $s \mathbin{\hat{}} t = 0_{[n]}$. Hence the equality is satisfiable if and only if $\mathcal{T}(s \mathbin{\hat{}} t, [0:n-1]) = F^1 \circ \ldots \circ F^m$ and $\bigwedge_{i=1}^{m} \neg F^i$ is satisfiable. At this point we can apply the technique used in [CMR97], which extract equalities from BDDs using equivalence preserving transformations of the form $\mathbf{ite}(x, H, G) \equiv (H \vee G) \ \wedge \ \exists\delta.x = H \wedge (\neg G \vee \delta)$. This produces a substitution θ_0 with subranges of the original variables in the domain and auxiliary δ's in the range. The resulting substitution can then be extracted by generating θ as follows:

$$\begin{array}{ll}
\theta_1: & [x \mapsto \theta_0(x_1) \circ \ldots \circ \theta_0(x_n) \mid x_i \in \mathbf{dom}(\theta_0) \ \wedge \ x = x_1 \circ \ldots \circ x_n] \\
\theta_2: & [x_k \mapsto x[i:j] \mid x = x_1 \circ \ldots \circ x_n, \ k \leq n, \ [i:j] = \mathit{decode}(x_k), \ \forall i : [1..n].x_i \notin \mathbf{dom}(\theta_0)] \\
\theta: & [x \mapsto \sigma(\theta_2(\theta_1(x))) \mid x \in \mathbf{dom}(\theta_1)]
\end{array}$$

Example: Continuing with the translated versions of our example terms s and t we will extract a substitution from the equality constraint $s = t$. We therefore complete the translation to get:

$$s \mathbin{\hat{}} t: \quad (w_1 \ \& \ y_3) \mathbin{\hat{}} (y_3 \mid x) \quad \circ \quad (w_2 \ \& \ y_4) \mathbin{\hat{}} (y_4 \mid 1_{[1]}) \quad \circ \quad (w_3 \ \& \ x) \mathbin{\hat{}} (y_2 \mid w_1)$$

By negating the concatenations, the constraints needed to extract a substitution are obtained. The second constraint is easiest as it simply imposes $w_2 = y_4 = 1_{[1]}$. The conjunction of the first and third constraint is transformed:

$$\begin{array}{ll}
 & \neg((w_1 \ \& \ y_3) \mathbin{\hat{}} (y_3 \mid x)) \ \& \ \neg((w_3 \ \& \ x) \mathbin{\hat{}} (y_2 \mid w_1)) = 1_{[3]} \\
\leftrightarrow & \mathbf{ite}(x, \ w_1 \ \& \ y_3 \ \& \ w_3, \ \neg y_2 \ \& \ \neg w_1 \ \& \ \neg y_3) = 1_{[3]} \\
\leftrightarrow & (x = w_1 \ \& \ y_3 \ \& \ w_3) \wedge ((w_1 \ \& \ y_3 \ \& \ w_3) \mid (\neg y_2 \ \& \ \neg w_1 \ \& \ \neg y_3)) = 1_{[3]} \\
\leftrightarrow & (x = w_1 \ \& \ y_3 \ \& \ w_3) \wedge (w_1 = y_3 \ \& \ w_3) \wedge ((y_3 \ \& \ w_3) \mid (\neg y_2 \ \& \ \neg y_3)) = 1_{[3]} \\
\leftrightarrow & (x = w_1 \ \& \ y_3 \ \& \ w_3) \wedge (w_1 = y_3 \ \& \ w_3) \wedge (y_3 = w_3) \wedge (w_3 \mid \neg y_2) = 1_{[3]} \\
\leftrightarrow & (x = w_1 \ \& \ y_3 \ \& \ w_3) \wedge (w_1 = y_3 \ \& \ w_3) \wedge (y_3 = w_3) \wedge \exists\delta.y_2 = w_3 \ \& \ \delta
\end{array}$$

The composition of the extracted equalities gives an idempotent substitution:

$$\theta : [w_1 \mapsto w_3 \ \& \ \delta, \quad x \mapsto w_3 \ \& \ \delta, \quad y_2 \mapsto w_3 \ \& \ \delta, \quad y_3 \mapsto w_3]$$

From this we generate a substitution, where $V_{aux} = \{w_3, \delta\}$.

$$\Big[x \mapsto w_3 \ \& \ \delta, \quad w \mapsto (w_3 \ \& \ \delta) \circ 1_{[1]} \circ w_3, \quad y \mapsto w_3 \circ 1_{[1]} \circ (w_3 \ \& \ \delta)\Big] \ .$$

7.3.2 Equational running time

For input $s = t_{[n]}$ not involving *op* subterms (basic bit-vectors) the presented algorithm can be tuned to run in time:

$$\mathcal{O}(m + n\log(n)),$$

where m is the number of $\circ$ and subfield extraction occurrences in s and t. First subfield extraction is pushed to the leaves in time $\mathcal{O}(m)$, then the $\circ$ subterms are arranged in a balanced tree and $\mathcal{T}$ is applied to the balanced terms while maintaining balance in the tree. The translated equality $s = t$ is processed in a style similar to *slice*, but the auxiliary function *apply* has been replaced by one that builds a graph by connecting vertices representing the equated constants or variables. If some connected component contains two different constants there is a contradiction and the equality is unsatisfiable. Otherwise an equivalence class representative is appointed for each connected component, choosing a constant if one is present, or an arbitrary variable vertex otherwise. The extracted substitution then maps every variable to a concatenation of equivalence class representatives.

A canonized solution for satisfiable equalities can be extracted in time $\mathcal{O}(n)$ (which is dominated by the running time of $\mathcal{T}$). An algorithm with the same functionality is presented in [CMR97]. That algorithm has running time $\mathcal{O}(m\log(m) + n^2)$, but offers some essential shortcuts that we don't address. Both procedures may still depend heavily on the parameter n. For instance, the equality

$$0_{[1]} \circ 1_{[1]} \circ x_{[m]} \quad = \quad x_{[m]} \circ 0_{[1]} \circ 1_{[1]} \tag{7.2}$$

requires (the maximal) m cuts of x, and is only satisfiable if m is even. The same functionality can, as [CMR97] noticed, be achieved in $\mathcal{O}(m + n)$ time, but at the expense of having this as the minimal running time as well.

Another advantage of our algorithm is that it can be extended (with a few modifications) to the case where bit-vectors of parameterized length are either exclusively on the right or exclusively on the left of every concatenation. This excludes cases like (7.2), which we will address in Section 7.4.

7.3.3 Beyond equalities

The satisfiability problem for constraints involving disequalities is NP-complete in the case of basic bit-vectors. Membership in NP follows from the fact that we can easily check in polynomial time that a given instantiation of bit-vector variables satisfies prescribed constraints. NP-hardness follows from a reduction from 3-SAT to conjunctions of disequality constraints: take an instance of 3-SAT $\bigwedge_i (l_i \vee k_i \vee m_i)$ where l_i, k_i and m_i are literals over the vocabulary $\mathcal{V}$ of boolean variables. Translate this into $\bigwedge_i (l_i \circ k_i \circ m_i \neq 000) \wedge \bigwedge_{x \in \mathcal{V}} (\overline{x} \neq x)$, where for each boolean variable x we associate two bit-vector variables $x_{[1]}$ representing x and $\overline{x}_{[1]}$ representing the negation of x.

We therefore settle here by handling $t \neq s$ as $|(t \hat{} s)$, and converting $|t_{[n]}$ to $t[0:0] \mid \ldots \mid t[n-1:n-1] = 1_{[1]}$. The connectives $<$ and $\leq$, as well as operations like $+$ and $*$

can be handled similarly, though the advantages of this approach are questionable. Naturally these constraints are only analyzed when all equational constraints have been processed and the resulting substitutions have been applied to the non-equational constraints.

Verification conditions of the form

$$f(A) \neq f(B) \ \wedge \ f(A) \neq f(C) \ \rightarrow \ \ f(B) = f(C),$$

where f is an uninterpreted function symbol, are handled using a complete case analysis on bit-vectors A, B and C (it is valid only when A, B and C are bit-vectors of length 1). Shostak's approach to combining equational theories misses cases like this as it is originally designed for theories admitting infinite models (see for example [NO79]).

7.4 Unification of basic bit-vectors

In this section we focus on the problem of finding unifiers for basic bit-vector terms s and t. The restriction to basic bit-vector terms allows us to develop a more efficient procedure and at the same time widen its scope to bit-vectors whose lengths are parameterized.

7.4.1 *ext*-terms

To more compactly represent solutions to equations like (7.2) we introduce a new bit-vector term construct $ext(t_{[n]}, m)$ (the extension of t up to length m), which is well-formed whenever $m > 0$. The meaning of ext is given by the equation

$$[\![ext(t_{[n]}, m)]\!]\eta \ \ = \ \ [\![\underbrace{t \circ .. \circ t}_{k} \circ t[0 : l{-}1]]\!]\eta \quad \text{where } (k{+}1)n \geq m > kn \text{ and } l = m{-}kn$$

Thus, $ext(t_{[n]}, m)$ repeats t up to the length m. To map ext-terms to terms in the base language we use the unfolding function unf

$$unf(t_{[n]}, m) \ \ = \ \ \underbrace{t \circ .. \circ t}_{k} \circ \mathcal{T}(t, [0 : l{-}1]) \quad \text{where } (k{+}1)n \geq m > kn \text{ and } l = m{-}kn$$

A solution to equation (7.2) can now be given compactly when m is even as $x = ext(0_{[1]} \circ 1_{[1]}, m)$.

7.4.2 Unification with *ext*-terms

To decide the satisfiability of equalities $s = t$ of basic bit-vector terms extended with ext-subterms we will develop a Martelli-Montanari style unification algorithm [MM82] which takes the singleton set $\mathcal{E}_0 : \{s = t\}$ as input and works by transforming $\mathcal{E}_0$ to intermediary sets $\mathcal{E}_1, \mathcal{E}_2, \ldots$ by equivalence preserving transformations which simplify, delete or propagate equalities. It ultimately produces either FAIL, when $s = t$ is unsatisfiable, or a substitution $\mathcal{E}_{final} : \{x_1 = t_1, \ldots, x_n = t_n\}$.

Since our procedure uses $\mathcal{T}$ to decompose terms, every auxiliary variable in $\mathcal{E}_{final}$ furthermore corresponds to a unique disjoint subrange of one of the original variables. The obviously satisfiable conjunction of equalities is equivalent to the original equality.

A canonizer can be obtained by first eliminating the *ext*-terms by using unfold and then using the canonizer of Section 7.3.1.

Example: Anticipating the algorithm we will present, consider the following equality assertion:

$$y_{[3]} \circ x_{[16]} \circ x_{[16]} \circ z_{[2]} = x_{[16]} \circ w_{[4]} \circ 0_{[1]} \circ x_{[16]} \ .$$

In processing the implied equality $y_{[3]} \circ x_{[16]} = x_{[16]} \circ \ldots$ we obtain $x_{[16]} = ext(y_{[3]}, 16)$ as a solution for $x_{[16]}$. Continuing with the remaining equalities we get the intermediate set of equations:

$$\begin{array}{lcllcl} x_{[16]} & = & ext(y_{[3]}, 16), & y_{[3]}[1:2] \circ y_{[3]}[0:0] & = & w_{[4]}[0:2], \\ z_{[2]} & = & w_{[4]}[3:3] \circ 0_{[1]}, & ext(w_{[4]}[3:3] \circ 0_{[1]}, 16) & = & x_{[16]} \ . \end{array}$$

The two equations involving x are combined to produce the implied constraint

$$ext(y_{[3]}, 16) = ext(w_{[4]}[3:3] \circ 0_{[1]}, 16) \ .$$

This equality is evidently equivalent to its *unf*-unfolding, but as we will later formulated in a general setting, we can do better and only need to assert:

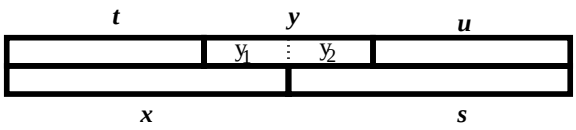
$$y_{[3]} \circ y_{[3]}[0:0] = w_{[4]}[3:3] \circ 0_{[1]} \circ w_{[4]}[3:3] \circ 0_{[1]} \ .$$

In fact this implies $y[0:0] = y[1:1] = y[2:2] = w[3:3] = 0_{[1]}$. After propagating the resulting constraints we obtain the final result:

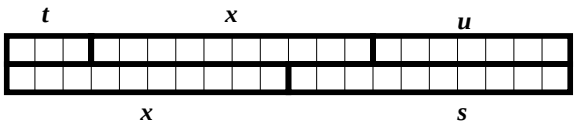
$$w_{[4]} = 0_{[4]}, \quad x_{[16]} = 0_{[16]}, \quad y_{[3]} = 0_{[3]}, \quad z_{[2]} = 0_{[2]} \ .$$ ☺

While the full unification algorithm is given in Figure 7.4 we highlight and explain the more delicate cases below.

$x_{[n]} \circ s = t_{[m]} \circ y_{[l]} \circ u$ when $m+l > n > m$, $x \neq y$. The situation is described in the picture below, which suggests that the equality is equivalent to $x = t \circ y_1$ and $s = y_2 \circ u$ for suitable splits y_1 and y_2 of y. We use $\mathcal{T}$ to cut y into the appropriate pieces. This replaces y everywhere in $\mathcal{E}$ by $y_1 \circ y_2$.



$x_{[n]} \circ s = t_{[m]} \circ x_{[n]} \circ u$ when $n > m$. For example we are given the configuration:



Thus, the original equality constraint is equivalent to $x = t \circ t \circ t \circ t[0:0]$ and $t[1:2] \circ t[0:0] \circ u = s$. To more compactly describe the first equality we use the *ext*-construct to obtain $x = ext(t, 10)$.

$ext(s_{[m]}, l) = ext(t_{[n]}, l)$ The effect of replacing x by s in the variable elimination step may introduce equality constraints between *ext*-terms. Although the equality constraint is by definition equivalent to $unf(s_{[m]}, l) = unf(t_{[n]}, l)$, we can be even more economical in the unfolding as the following lemma suggests.

Lemma 7.4.1 *Assume* $l \geq n + m - \gcd(m, n)$ *and let* $g = \gcd(m, n)$ *then*

$$ext(s_{[m]}, l) = ext(t_{[n]}, l) \quad \leftrightarrow \quad unf(s, m + n - g) = unf(t, m + n - g)$$

Proof:

First divide s and t into slices each of size g and let $p = \frac{m}{g}$ and $q = \frac{n}{g}$. Assume without loss of generality that $p < q$. With s is divided into p pieces and t into q pieces, $unf(s, m + n - g) = unf(t, m + n - g)$ now creates $p + q - 1$ equalities between slices from s and slices from t. The assertion is no stronger than the original equality as the assumptions of the lemma guarantee that $m + n - g < l$. The bipartite graph associated with these slices and equalities has $p + q$ vertices and $p + q - 1$ edges.

If the graph had a cycle of length $2k$, then $k < p$ as only the $p - 1$ first t slices are repeated. Since the hypothetical cycle starts and ends at the same position in s and two consecutive vertices of the cycle are in distance $q \bmod p$ of eachother it would also imply that p divides $k \cdot (q \bmod p)$. But then since p and q are relatively prime $1 = \gcd(p, q) = \gcd(p, q \bmod p)$ and so p divides k which is impossible.

Hence, the bipartite graph is a spanning tree and all s slices are equated with all t slices. It is therefore sufficient to equate the unfoldings of s and t up to $m + n - g$ as the effect of unfolding is unchanged from this point on. ∎

Thus, we will ensure that our algorithm maintains the invariant $2n \leq l$ for every $ext(t_{[n]}, l)$ term, and the equality constraint $ext(s_{[m]}, l) = ext(t_{[n]}, l)$ is replaced by $unf(s, m + n - g) = unf(t, m + n - g)$.

Other simpler cases are summarized in Figure 7.4. It omits cases that can be obtained using symmetry of equality.

Constructor elimination

R1 $\{s_{[m]} \circ u = t_{[m]} \circ v\} \cup \mathcal{E} \rightarrow \{s = t, u = v\} \cup \mathcal{E}$

R2 $\{c_{[m]} \circ s = c'_{[n]} \circ t\} \cup \mathcal{E} \rightarrow$ FAIL **where** $c \neq c'$

R3 $\{c_{[m]} \circ s = c_{[n]} \circ t\} \cup \mathcal{E} \rightarrow \{s = c_{[n-m]} \circ t\} \cup \mathcal{E}$ **where** $n > m$

R4 $\{x_{[n]} \circ s = t_{[m]} \circ y_{[l]} \circ u\} \cup \mathcal{E} \rightarrow \{x = t \circ y_1, s = y_2 \circ u\} \cup \mathcal{E}$
where $m + l > n > m > 0$, $x \neq y$,
$y_1 = \mathcal{T}(y, [0 : m - n - 1])$, $y_2 = \mathcal{T}(y, [m - n : l - 1])$,

R5 $\{x_{[n]} \circ s = t_{[m]} \circ x_{[n]} \circ u\} \cup \mathcal{E} \rightarrow \{s = \mathit{wrap}(t, n) \circ u, x = \mathit{mk\text{-}ext}(t, n)\} \cup \mathcal{E}$
where $n > m > 0$

R6 $\{x_{[n]} \circ s = t_{[m]} \circ c_{[l]} \circ u\} \cup \mathcal{E} \rightarrow \{x = t \circ c_{[m-n]}, s = c_{[l+n-m]} \circ u\} \cup \mathcal{E}$
where $m + l > n > m \geq 0$

R7 $\{s_{[m]} \circ t = \mathit{ext}(u_{[l]}, n) \circ v\} \cup \mathcal{E} \rightarrow \left\{ \begin{array}{l} s_{[m]} = \mathit{mk\text{-}ext}(u, m), \\ t = \mathit{mk\text{-}ext}(\mathit{wrap}(u, m), n - m) \circ v \end{array} \right\} \cup \mathcal{E}$
where $m < n$

R8 $\{\mathit{ext}(s_{[l_1]}, m) = \mathit{ext}(t_{[l_2]}, m)\} \cup \mathcal{E} \rightarrow \{\mathit{unf}(s, l) = \mathit{unf}(t, l)\} \cup \mathcal{E}$
where $l = l_1 + l_2 - \gcd(l_1, l_2)$

Equality and variable elimination

R9 $\{t = t\} \cup \mathcal{E} \rightarrow \mathcal{E}$

R10 $\{x = s\} \cup \mathcal{E} \rightarrow \{x = s\} \cup \mathcal{E}[x \mapsto s]$

Figure 7.4: Rules for unification with *ext*-terms

The auxiliary function *wrap* splits the term t at position k and swaps the two pieces. The function *mk-ext* produces either an *ext*-term when the length of t is sufficiently small or unfolds t. It ensures that every $\mathit{ext}(t_{[n]}, m)$ term generated by the algorithm satisfies $2n \leq m$. These are defined more precisely below:

$$\mathit{wrap}(t_{[n]}, m) = \textbf{let}\ \ k = m \bmod n\ \textbf{in}$$
$$\textbf{if}\ \ k = 0\ \textbf{then}\ t\ \textbf{else}\ \mathcal{T}(t, [k : n - 1]) \circ \mathcal{T}(t, [0 : k - 1])$$

$$\mathit{mk\text{-}ext}(t_{[n]}, m) = \textbf{if}\ \ 2n \leq m\ \textbf{then}\ \ \mathit{ext}(t, m)\ \textbf{else}\ \ \mathit{unf}(t, m)$$

The unification algorithm terminates since the variable elimination step removes duplicate constraints involving x and every other step produces equalities of smaller size (in terms of the number of bitwise comparisons) than the one eliminated. For instance, in the R8 rule we rely on $m \geq 2 \cdot \max(l_1, l_2) > l$.

7.4.3 Nonfixed size bit-vectors

The most prominent feature of the unification algorithm in Figure 7.4 is that it can be used to decide bit-vector equality constraints $s = t$, where lengths and projections are not

restricted to fixed naturals, but are of the form $aN + b$, where a and b are integers and N is a parameter (where we assume without loss of generality that $N > 0$). This allows us to apply the algorithm in the Shostak combination for deciding verification conditions with non-fixed bit-vector equalities. The unification problem for non-fixed bit-vectors is also reminiscent of the word unification problem, see Section 8.2.2. The main difference with word unification is that variables ranging over words in that problem do not have associated size constraints which bit-vectors have. By performing comparisons and arithmetic on these lengths symbolically and allowing admissible answers to be paired with accumulated constraints (as explained later), we can deal with the following example:

Example: By performing the unification of

$$\{w_{[2]} \circ 0_{[1]} \circ x_{[N+6]} \circ y_{[N+7]} = x_{[N+6]} \circ 1_{[1]} \circ z_{[3]} \circ x_{[N+6]}\} \tag{7.3}$$

we obtain as an intermediate step

$$\left\{ \begin{array}{c} x_{[N+6]} = ext(w_{[2]} \circ 0_{[1]}, N+6), \\ y_{[N+7]} = z_{[3]}[2:2] \circ x_{[N+6]} \\ 1_{[1]} \circ z_{[3]}[0:1] = wrap(w_{[2]} \circ 0_{[1]}, N+6) \end{array} \right\}$$

and finally two cases:

$$\begin{array}{lcll} x_{[N+6]} & = & ext(1_{[1]} \circ \beta_{[1]} \circ 0_{[1]}, N+6), & \\ y_{[N+7]} & = & \alpha_{[1]} \circ ext(1_{[1]} \circ \beta_{[1]} \circ 0_{[1]}, N+6), & \text{if } N \equiv 0 \pmod 3 \\ z_{[3]} & = & \beta_{[1]} \circ 0_{[1]} \circ \alpha_{[1]}, & \\ w_{[2]} & = & 1_{[1]} \circ \beta_{[1]} & \end{array}$$

$$\begin{array}{lcll} x_{[N+6]} & = & ext(\beta_{[1]} \circ 1_{[1]} \circ 0_{[1]}, N+6), & \\ y_{[N+7]} & = & \alpha_{[1]} \circ ext(\beta_{[1]} \circ 1_{[1]} \circ 0_{[1]}, N+6), & \text{if } N \equiv 1 \pmod 3 \\ z_{[3]} & = & 0_{[1]} \circ \beta_{[1]} \circ \alpha_{[1]}, & \\ w_{[2]} & = & \beta_{[1]} \circ 1_{[1]} & \end{array}$$

When $N \equiv 0 \pmod 3$, the evaluation of the *wrap* function simplifies the second equation of the intermediate result to $1_{[1]} \circ z_{[3]}[0:1] = w_{[2]} \circ 0_{[1]}$. The case that corresponds to $N \equiv 2 \pmod 3$ requires $1_{[1]} \circ z_{[3]}[0:1] = 0_{[1]} \circ w_{[2]}$ which results in an inconsistency. The $\beta_{[1]}$,$\alpha_{[1]}$ are auxiliary variables that are introduced to represent unknown segments of the bit-vector variables.

Thus, the result produced by the unification algorithm will now be a *set* of constraints, each of the form

$$(ax + b > c, [N \mapsto ax + b] \circ [x_i \mapsto t_i \mid i = 1, \ldots, n])$$

where x is a fresh variable and the first constraint is passed on to decision procedures for linear arithmetic, and the second constraint is a substitution. We are thus faced with a finitary as opposed to unitary unification problem (see [BS93] for a survey on unification theory).

The crucial observation that allows lifting the algorithm to parameterized bit-vector expressions is that all operations and tests on the lengths and projections are of the form

$$m+n, \quad m-n, \quad m>n, \quad m \geq n, \quad m=n, \quad m \bmod n.$$

Since terms of the form $aN+b$ are closed under addition and subtraction, the first two operations can be performed directly in a symbolic way.

The comparison $m > n$ is rewritten to $m-n>0$, $m \geq n$ to $m-n+1>0$, and $n=m$ to $n-m+1>0 \ \wedge \ m-n+1>0$. This reduces the evaluation of comparisons to $aN+b>0$. Since

$$\begin{aligned} aN+b>0 \ \leftrightarrow \ (a=0 \wedge b>0 \ \vee \ a>0) \ \text{ iff} \\ (a>0>b \vee a<0<b) \ \rightarrow N \geq |b| \text{ div } |a| \end{aligned} \tag{7.4}$$

tests can be evaluated using $a=0 \wedge b>0 \ \vee \ a>0$ and accumulating auxiliary lower bounds on N for a separate treatment. Our algorithm then produces answers for all N greater than the largest accumulated lower bound. For values of N smaller than the accumulated bounds we instantiate N and run the fixed size version.

The auxiliary function *wrap* requires us to compute $m \bmod n$. To simplify this case our algorithm will maintain the invariant that $m \bmod n$ is only invoked when n is a constant b', whereas m may be of the form $N+b$. The case $N \geq b'-b$ causes case-splitting on each of the possible solutions $k=0,\ldots,b'-1$.

We could represent each case in Presburger arithmetic as $\exists x \geq 0 \ . \ xb' = N+b-k$ and use a Presburger decision procedure [Coo72] to check satisfiability of conjunctions of such constraints. However, in order to manage these constraints more efficiently we can use the Chinese Remainder Theorem (see [NZM91]). If $\Pi_i p_i^{\alpha_i}$ is a prime factorization of b' (with $p_1, p_2, \ldots$ the sequence of all primes), then

$$N+b \equiv k \pmod{b'} \quad \text{iff} \quad N+b \equiv k \pmod{p_i^{\alpha_i}} \text{ for every } i.$$

Let $D(p,\beta,l)$ be the predicate that $N \equiv l \pmod{p^\beta}$ is true. Let $\mathcal{C}_{\text{mod}} = \bigwedge_i D(p_i,\beta_i,b_i)$ be the conjunction of divisibility constraints imposed on the current system. Only one predicate is needed for each p_i, since:

$$D(p,\beta,l') \wedge D(p,\alpha,l) \wedge \beta \geq \alpha \quad \text{iff} \quad D(p,\beta,l') \wedge l' \equiv l \pmod{p^\alpha} \ . \tag{7.5}$$

In order to split on the case $N+b \equiv k \pmod{b'}$ for different values of $k=0,\ldots,(b'-1)$ we can form the product of the case splits on $N+b \equiv k_i \pmod{p_i^{\alpha_i}}$ for $k_i=0,\ldots,(p_i^{\alpha_i}-1)$ (the product is over $i=1,2,\ldots$). The situation is not as bad as it seems, since we can use

the existing $\mathcal{C}_{\text{mod}}$ to merge the new constraints in an optimal way:

$$\mathcal{C}'_{\text{mod}} = \bigwedge_i P(i) \quad \text{where} \quad P(i) = \begin{cases} \bigvee_{j=0}^{p_i^{\alpha_i-\beta_i}-1} D(p_i, \alpha_i, b_i + jp_i^{\beta_i}) & \text{if } \alpha_i \geq \beta_i \\ D(p_i, \beta_i, b_i) & \text{if } \alpha_i < \beta_i \end{cases}$$

The predicate $P(i)$ represents the enumeration of valid congruences modulo a power of p_i. Statement (7.5) suggests the form of the enumeration for each case in the definition of $P(i)$. Expressing $\mathcal{C}'_{\text{mod}}$ in disjunctive normal form $\bigvee_i \mathcal{C}^i_{\text{mod}}$ the constraints for the different cases are obtained. The value of k for a particular case of $\mathcal{C}_{\text{mod}}$ can be reconstructed using the congruence

$$k \equiv \left(\sum_i n_i b_i\right) - b \pmod{b'}$$

where $n_i = z_i \overline{z}_i$, $z_i = \Pi_{j \neq i} p_j^{\alpha_i}$, and $\overline{z}_i$ satisfies $z_i \overline{z}_i \equiv 1 \pmod{p_i^{\alpha_i}}$ (it exists since $\gcd(p_i^{\alpha_i}, z_i) = 1$).

Given expressions s and t our algorithm now engages in the following steps:

1. Apply $\mathcal{T}$ to both s and t, i.e., let $(s,t) := (\mathcal{T}(s,[0:m-1]), \mathcal{T}(t,[0:m-1]))$. This generates bit-vector expressions without subfield extraction and an assignment to each original variable x to a concatenation $x_1 \circ x_2 \circ \ldots \circ x_n$ of distinct variables, where $\mathit{decode}(x_i)$ cover disjoint intervals of x. Using equivalence (7.4) the tests in $\mathcal{T}$ are evaluated unambiguously, and possibly generating a new lower bound on N. The cases where N is smaller than this bound are processed later.

2. Every variable $x_{[aN+b]}$ remaining in s or t, where $a > 0$, is replaced by a concatenation of a fresh variables: $x^{(1)}_{[N]} \circ x^{(2)}_{[N]} \circ \ldots \circ x^{(a)}_{[N+b]}$. Constants are cut in a similar way[1]. If b is negative the lower bound $1 - b$ on N is added.

   Every variable occurring in s and t now has length $N + k$ or k, where k is an integer.

3. The algorithm in Figure 7.4 is invoked on the equality $\{s = t\}$. Each comparison accumulates a lower bound on N and each invocation of mod may cause a multi-way case split while accumulating modulus constraints on N. The unification algorithm therefore generates constraints of the form $(\mathcal{E}_1, \mathcal{C}_1), \ldots, (\mathcal{E}_n, \mathcal{C}_n)$, where the $\mathcal{E}_i$ are equalities and $\mathcal{C}_i$ is a conjunction of $N \geq k$ and $D(p_i, \alpha_i, a_i)$ constraints.

   We need to ensure that every step is well defined: in particular that $\mathit{unf}(t,m)$ and, as we assumed, n mod m are only invoked when m is a constant. This is a consequence of the following invariant:

   Invariant 7.4.2 *For every occurrence of* $\mathit{ext}(t_{[aN+b]}, n)$*:* $a = 0 \wedge 2b \leq n$.

[1] This step is not strictly necessary, but simplifies the further presentation of the algorithm.

This holds as *ext* terms are only generated when $mk\text{-}ext(t_{[aN+b]}, a'N + b')$ is invoked and $2(aN + b) \leq a'N + b'$. Since both a and a' are either 0 or 1, this inequality can only hold if $a = 0$ or N is bounded above by $(b' - 2b)$ div $(2a - a')$. The cases where N is bounded above by a constant are treated separately.

4. The solved form can now be extracted. For each $(\mathcal{E}, \mathcal{C})$ generated from the previous step let $\mathcal{C}$ be of the form $N \geq k \wedge \bigwedge_{i=1}^{l} D(p_i, \alpha_i, a_i)$. The Chinese Remainder Theorem tells us how to find n_i such that the constraints can be rewritten to the equivalent form

$$N \geq k \wedge \exists x.N = Ax + B \quad \text{where} \quad A = \prod_{i=1}^{l} p_i^{\alpha_i} \quad B = \left(\sum_{i=1}^{l} n_i a_i\right) \bmod A$$

Since we extract the Shostak substitution θ from $\mathcal{E}$ as in the fixed-length case the combined constraint returned for this case is

$$(Ax + B \geq k, [N \mapsto Ax + B] \circ \theta).$$

For each k less than the least lower bound accumulated above we instantiate N by k and extract θ_k by running the fixed-size version of the algorithm (that is, running $\{s = t\}[N \mapsto k]$). For these cases the returned constraints have the form

$$(\texttt{true}, [N \mapsto k] \circ \theta_k).$$

The algorithm now concludes by returning the entire set of the constraints extracted above.

As we have argued above we now have

Theorem 7.4.3 *(Correctness) When the non-fixed unification algorithm terminates on the input constraint $s = t$ with a set of constraints $\{(\varphi_i(x), \theta_i) \mid i = 0, \ldots n\}$ then $s = t \quad \leftrightarrow \bigvee_{i=0}^{n} \exists x, V_{aux}.\varphi_i(x) \wedge \theta_i$.*

Finally we must ensure that we can make the unification algorithm modified for parameterized lengths terminate. To this end we apply the transformation rules from Figure 7.4 by preferring the variable and equality elimination rules to the other rules.

We will proceed to prove the termination by induction on the number of distinct non-fixed variables k in $\mathcal{E}$ that participate in some equality where rule R1-R8 can be applied. The base case ($k = 0$) operates only on fixed-size variables, and so it terminates.

Whenever a variable x has been isolated using one of the rules R4-R6, it is eliminated from the rest of $\mathcal{E}$. Indeed it is eliminated as x cannot be a proper subterm of t in the equality constraint $x = t$, since the length of t is the sum of the lengths of its variable and constant subterms, which equals the length of x. Since rules R1-R8 produce equalities between smaller bit-vectors we cannot repeatedly apply these rules without eventually eliminating a non-fixed size variable. Rule R4 may split a non-fixed length variable y into two parts

1.	equation (7.3) from page 127	satisfiable	0.06 s
2.	$0_{[1]} \circ 1_{[1]} \circ 0_{[1]} \circ x_{[N+7]} \circ 1_{[1]} \circ 0_{[1]} \circ 1_{[1]} \circ y_{[N+1]}$ $= x_{[N+7]} \circ x_{[N+7]}$	unsatisfiable	0.06 s
3.	$x_{[N+4]} \circ 0_{[1]} \circ 1_{[1]} \circ 0_{[1]} \circ y_{[N+9]}$ $= y_{[N+9]} \circ 1_{[1]} \circ 0_{[1]} \circ 1_{[1]} \circ x_{[N+4]}$	unsatisfiable	0.09 s
4.	$(7.3) \rightarrow z_{[3]}[0:0] = 0_{[1]} \vee z_{[3]}[1:1] = 0_{[1]}$	valid	0.07 s

Table 7.1: Non-fixed bit-vectors examples

y_1 and y_2, but only one of these parts will have non-fixed length, so the overall number of non-fixed length variables is constant.

We therefore have

Theorem 7.4.4 *(Termination) The non-fixed unification algorithm terminates.*

A reduction from the problem of simultaneous incongruences [SM73] can establish that the unification problem for non-fixed bit-vectors is NP-hard. As it is formulated in Garey and Johnson [GJ79]: given a collection $\{(a_1, b_1), \ldots, (a_n, b_n)\}$ of ordered pairs of positive integers with $a_i \leq b_i$ for $1 \leq i \leq n$ the question whether there is an integer N such that for $q \leq i \leq n$, $N \not\equiv a_i \pmod{b_i}$ is NP-complete. This problem is reduced to basic bit-vector unification using auxiliary bit-vector variables x^i of length $2b_i - 1$ and y^i of length $2N - 2a_i + 2b_i - 1$ for $1 \leq i \leq n$. (the factor 2 is used to guarantee that all lengths are positive). Now N exists if and only if the equations

$$ext(x^i \circ 1_{[1]}, 2N - 2a_i + 2b_i) = y^i \circ 0_{[1]}$$

are satisfiable (can be unified). Naturally the n equalities can be combined to form one equality by concatenating all left-hand sides and all right-hand sides. On the other hand, a simple analysis of the termination argument can establish that a satisfying unifier can be verified in time polynomial in the constant parameter sizes and number of subterms.

The unification algorithm finally needs to be supplied also with a canonizer that works on *ext*-terms of non-fixed length to enable an integration with other decision procedures. While simple unfoldings cannot be performed this time our implementation normalizes terms into a concatenation of variables, constants and *ext*-terms whose arguments are fixed size terms in canonical form. The occurrences of *ext* in the resulting expression are then shifted as much as possible to the left. This step cannot be performed unambiguously without asserting congruence constraints on the parameter and hence also leads to case splits.

Table 7.1 gives a list of examples that were presented to our prototype implementation. The tests were made on a 200MHz Sun Ultra II.

7.5 Problems

Problem 7.5.1 *Integrate arithmetical reasoning "efficiently" with bit-wise boolean operations.*

Problem 7.5.2 *Extend non-fixed solving to bit-wise boolean operations.*

Problem 7.5.3 *Solve arithmetical constraints over the p-adics instead of over field of fixed size (say* 2^{32}*).*

Problem 7.5.4 *Find a terminating solver for non-fixed bit-vector constraints for bit-vectors whose lengths are given in quantifier-free Presburger (integer linear) arithmetic.*

7.6 Summary

This chapter presented two algorithms: one algorithm handles boolean operations on fixed-size bit-vectors, the other handles equational constraints in the absence of boolean operations on parameterized bit-vectors. A completed picture would combine the algorithms to handle boolean operations on parameterized bit-vectors.

Chapter 8

Queues

This chapter offers a solver-oriented decision procedure for queues. We first solve equational constraints. In analogy with recursive data-types we also develop decision methods for queue prefix, suffix, and sub-queue relations. We first motivate the decision procedures for queues with a small example.

8.1 Verification with queues

A generic situation for network routers and controllers whose input is a sequence of bits, is to congest the bit sequence in some way for a consumer. Take for instance the situation where a random sequence of bits has to be ordered in equal valued chunks of length $N > 0$ to the consumer. After the router has emitted N bits of the same value it is required to emit the other value, but it may only emit bits that have been received. Although seemingly artificial, this very scenario has been used to model a traffic controller along the Californian coast in [Bjø98a].

It should be noted that linear time temporal logic provides a convenient formalism for capturing more precise requirements of the router. In this chapter we will only concentrate on how queues are used to model the protocol and how decision procedures are used to automatically establish properties for queues. We will not discuss how temporal requirements may be captured for this example, but proceed to present a sample implementation directly. Figure 8.1 suggests an implementation of such a router. It uses a stack to keep track of bits that cannot be sent immediately, a counter i to maintain how many bits of the same value have been sent, and a flag *turn* to record whose turn it is. The use of a stack allows to adapt the implementation to the case where the bits are replaced by records where only one of the fields contains the bits used in this simplification. The asynchronous channels *producer* and *consumer* are modeled using queues, such that the statement

$$consumer \Longleftarrow v$$

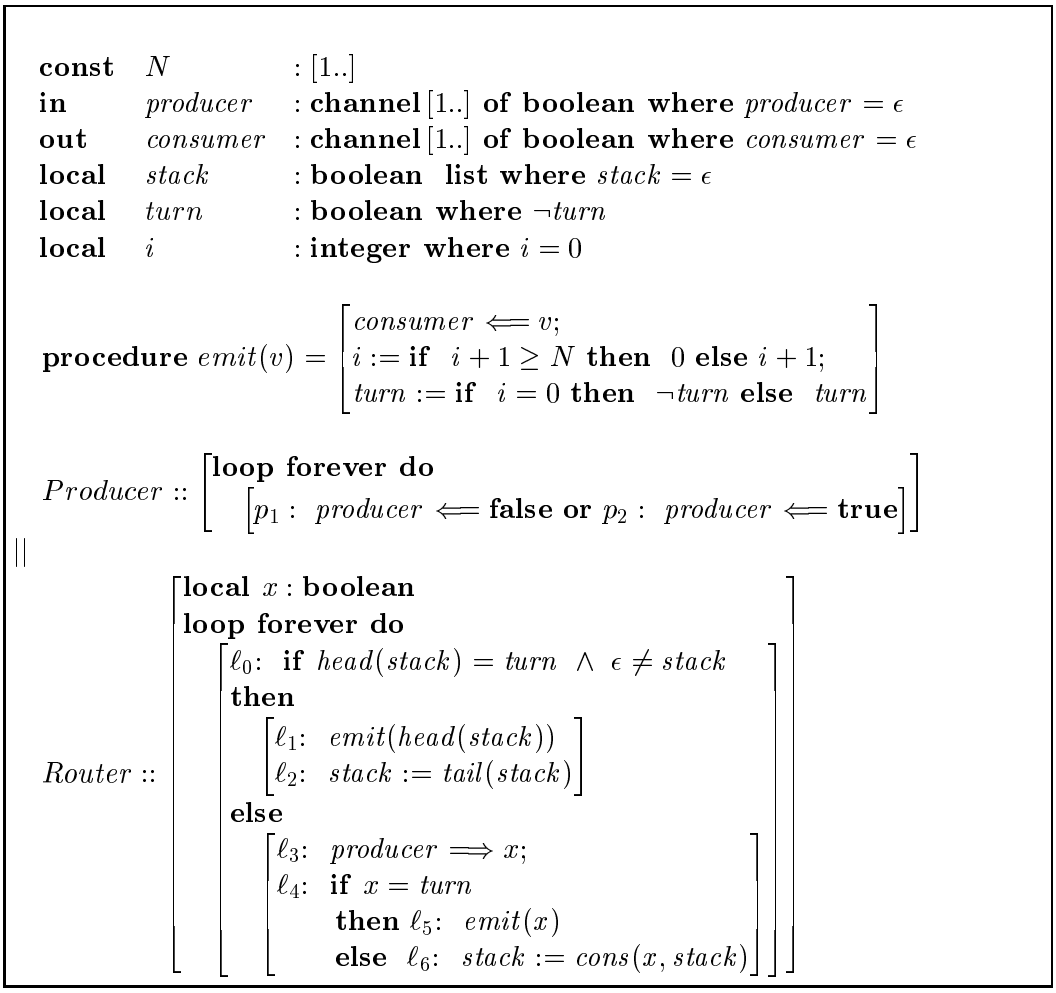


Figure 8.1: Program ROUTER

has as effect to put v in the end of *consumer*. The statement

$$producer \Longrightarrow x$$

can be executed when *producer* is non-empty, and has the effect of dequeueing the first element from *producer* and updating x to it.

It is a property of the implementation that the *stack* variable contains only bits of the same value. We can check this by postulating the invariant:

$$(\neg head(stack)) \notin stack \tag{8.1}$$

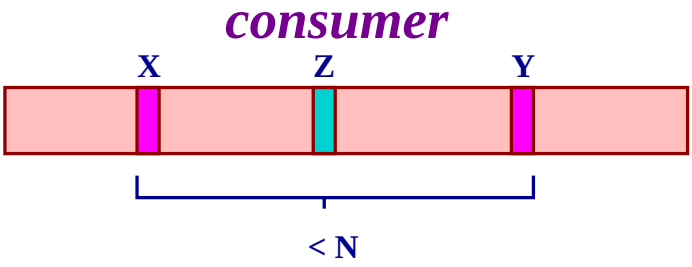
The invariant is not inductive, but it is possible to use the automatically generated local invariants:

$$at\_\ell_{3,4} \ \wedge \ (head(stack) \leftrightarrow turn) \Rightarrow stack = \epsilon \tag{8.2}$$

$$at\_\ell_{5,6} \ \Rightarrow \ (x \leftrightarrow turn) \wedge ((head(stack) \leftrightarrow turn) \rightarrow stack = \epsilon) \tag{8.3}$$

We can use rule INV from Figure 1.2 and the decision procedures presented later in this chapter to automatically prove the property.

Suppose now that we wish to express that the bits in the consumer do not change value within distance N. Pictorially, if x and y are the same in *consumer*, and the distance between x and y does not exceed N, then any z between x and y must have the same value.



Using a sub-queue relation symbol $\preceq$, operations *head*, and *last*, which pick first and last elements in a queue, and a length measure $|\ |$ we can express this consisely using the invariant:

$$(\forall s)\left(\left(\begin{array}{ll} & s \preceq consumer \\ \wedge & 1 \leq |s| \leq N \\ \wedge & head(s) = last(s) \end{array}\right) \Rightarrow \ (\neg head(s)) \notin s\right) \tag{8.4}$$

The invariant is unfortunately not inductive, but can be established using the auxiliary invariants below. The predicate *suffix* states that s is a suffix of the queue *consumer*.

$$\Box(0 \leq i < N) \tag{8.5}$$

$$i > 0 \quad \Rightarrow \quad last(consumer) = turn \tag{8.6}$$

$$i = 0 \ \wedge \ consumer \neq \epsilon \quad \Rightarrow \quad last(consumer) \neq turn \tag{8.7}$$

$$(\forall s) \left(\begin{array}{l} \left(\begin{array}{l} suffix(s, consumer) \ \wedge \ \ 1 \leq |s| \leq N \end{array} \right) \\ \qquad \Rightarrow \ \textbf{if} \ \ |s| \leq i \ \textbf{then} \ \ \neg turn \notin s \ \textbf{else} \ \ (i = 0 \ \rightarrow \ \ turn \notin s) \end{array} \right) \tag{8.8}$$

The good news is on the other hand that verification of both the auxiliary invariants and the main specification proceeds practically automatically thanks to the decision procedures for queues that we develop in the following. The verification condition below is one of the proof-obligations that is established in 22 seconds using the decision procedures.

$$\left(\begin{array}{ll} & (0 \leq i \ \wedge \ i < N) \\ \wedge & (0 < i \rightarrow \texttt{last}(consumer) = turn) \\ \wedge & \left(\begin{array}{l} i = 0 \ \wedge \ \neg(consumer = \epsilon) \rightarrow \\ \quad \neg(\texttt{last}(consumer) = turn) \end{array} \right) \\ \wedge & \left(\begin{array}{l} \mathsf{suffix}(\texttt{first}(s), consumer) \\ \ \wedge 1 \leq |\texttt{first}(s)| \ \wedge \ |\texttt{first}(s)| \leq N \\ \quad \rightarrow \\ \quad \textbf{if} \ |\texttt{first}(s)| \leq i \\ \qquad \textbf{then} \ (\neg turn) \notin \texttt{first}(s) \\ \qquad \textbf{else} \ (i = 0 \rightarrow turn \notin \texttt{first}(s)) \end{array} \right) \\ \wedge & \left(\begin{array}{l} s \preceq consumer \ \wedge \ 1 \leq |s| \ \wedge \ |s| \leq N \rightarrow \\ \quad \texttt{head}(s) = \texttt{last}(s) \rightarrow (\neg \texttt{head}(s)) \notin s \end{array} \right) \\ \wedge & \texttt{head}(s) = \texttt{last}(s) \\ \wedge & s \preceq \texttt{revcons}(consumer, turn) \\ \wedge & 1 \leq |s| \ \wedge \ |s| \leq N \\ \wedge & \left(\begin{array}{l} \texttt{first}(s) \preceq consumer \ \wedge \\ 1 \leq |\texttt{first}(s)| \ \wedge \ |\texttt{first}(s)| \leq N \rightarrow \\ \texttt{head}(\texttt{first}(s)) = \texttt{last}(\texttt{first}(s)) \rightarrow \\ (\neg \texttt{head}(\texttt{first}(s))) \notin \texttt{first}(s) \end{array} \right) \end{array} \right) \rightarrow (\neg \texttt{head}(s)) \notin s$$

Finally, we can verify that elements in distance $N + 1$ in the *consumer* are always different using the auxiliary invariant (8.9) in establishing (8.10).

$$(\forall s) \, (suffix(s, consumer) \ \wedge i < |s| \leq N \ \Rightarrow \ \texttt{head}(s) \neq \texttt{last}(s)) \tag{8.9}$$

$$(\forall s) \, (s \preceq consumer \ \wedge \ |s| = N + 1 \ \Rightarrow \ \texttt{head}(s) \neq \texttt{last}(s)) \tag{8.10}$$

8.2 A theory of queues

We use the sort S queue to refer to queues over the base sort S, and admit the following operations and relations:

$$\begin{array}{rl} \epsilon: & S \text{ queue}, \\ \texttt{head}: & S \text{ queue} \to S, \\ \texttt{tail}: & S \text{ queue} \to S \text{ queue}, \\ \texttt{last}: & S \text{ queue} \to S, \\ \texttt{first}: & S \text{ queue} \to S \text{ queue}, \\ \texttt{revcons}: & S \text{ queue} \times S \to S \text{ queue}, \\ \texttt{cons}: & S \times S \text{ queue} \to S \text{ queue} \end{array} \qquad \begin{array}{rl} \doteq: & S \text{ queue} \times S \text{ queue} \to \mathcal{B} \\ \mathsf{prefix}: & S \text{ queue} \times S \text{ queue} \to \mathcal{B} \\ \mathsf{suffix}: & S \text{ queue} \times S \text{ queue} \to \mathcal{B} \\ \preceq: & S \text{ queue} \times S \text{ queue} \to \mathcal{B} \\ \in: & S \text{ queue} \times S \text{ queue} \to \mathcal{B} \end{array}$$

The empty queue is written ϵ, and the usual list operations, `head`, `tail` and `cons` are supplemented with dual operations `last`, `first`, and `revcons`. The effect of the constructors and selectors is summarized in Figure 8.2. Thus, if x is not ϵ, and $a = \texttt{head}(x)$, $y = \texttt{tail}(x)$, then $x = \texttt{cons}(a, y)$, and symmetrically for the operators `revcons`, `first`, and `last`. Figure 8.3 summarizes the first-order theory of the queue selectors and constructors. The operations are supplemented by the equality relations, as well as the binary relations prefix, suffix, $\preceq$, and $\in$. We write $\mathsf{prefix}(x, y)$ if x is a prefix of y, $\mathsf{suffix}(x, y)$ if x is a suffix of y, and $x \preceq y$ if x is a subsequence of y. Taking $\circ$ as the concatenation of sequences we can define these relations using

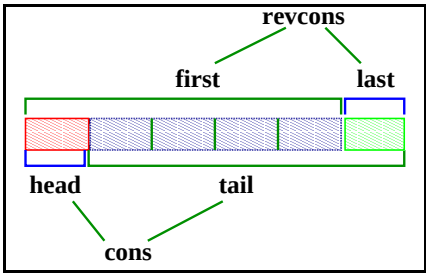


Figure 8.2: Queue constructors and selectors

$$\mathsf{prefix}(x, y) \stackrel{\text{def}}{=} \exists z \,.\, x \circ z = y \qquad \mathsf{suffix}(x, y) \stackrel{\text{def}}{=} \exists z \,.\, z \circ x = y$$

$$x \preceq y \stackrel{\text{def}}{=} \exists z, u \,.\, z \circ x \circ u = y \qquad a \in y \stackrel{\text{def}}{=} [a] \preceq y$$

where $[a]$ is shorthand for $\texttt{cons}(a, \epsilon)$.

The decision procedures that we develop here will for instance be able to establish validity of formulas such as

$$q \neq \epsilon \;\to\; q \doteq \texttt{cons}(\texttt{head}(q), \texttt{tail}(q)) \tag{8.11}$$

$$q \neq \epsilon \;\to\; \texttt{head}(\texttt{revcons}(q, a)) \doteq \texttt{head}(q) \tag{8.12}$$

$$a \notin q \;\wedge\; br \preceq q \;\to\; b \neq a \tag{8.13}$$

8.2.1 First-order decision procedures

In the full first-order theory of queues we can eliminate selectors completely by introducing four fresh constants: $a_{\texttt{head}}$, $q_{\texttt{tail}}$, $a_{\texttt{last}}$, $q_{\texttt{first}}$, and replacing subformulas with selectors

For all $a, b : S$, $x, y : S$ queue	
selectors	$\texttt{first}(\texttt{revcons}(x, a)) = x$ $\quad$ $\texttt{last}(\texttt{revcons}(x, a)) = a$
	$\texttt{head}(\texttt{cons}(a, x)) = a$ $\quad$ $\texttt{tail}(\texttt{cons}(a, x)) = x$
constructors	$\texttt{revcons}(x, a) = \texttt{revcons}(y, b) \ \rightarrow \ x = y \ \wedge \ a = b$
	$\texttt{cons}(a, x) = \texttt{cons}(b, y) \ \rightarrow \ a = b \ \wedge \ x = y$
exchange	$\texttt{revcons}(\epsilon, a) = \texttt{cons}(a, \epsilon)$
	$\texttt{revcons}(\texttt{cons}(a, x), b) = \texttt{cons}(a, \texttt{revcons}(x, b))$
acyclicity	$\texttt{revcons}(x, a) \neq \epsilon$ $\quad$ $\texttt{cons}(a, x) \neq \epsilon$
$x \neq \texttt{revcons}(..\texttt{revcons}(\texttt{cons}(a_1, .., \texttt{cons}(a_n, x)..), b_1), .., b_m); \quad n + m > 0$	
domain closure	$x = \epsilon \ \vee \ \exists a : S, y : S \ \texttt{queue} \ . \ x = \texttt{cons}(a, y)$

Figure 8.3: Equational axioms for queue operations

using transformations of the form:

$$\varphi[\texttt{head}(t)] \quad \mapsto \quad t = \epsilon \ \wedge \ \varphi[a_{\texttt{head}}] \ \vee \ \exists a, x \ . \ t = \texttt{cons}(a, x) \ \wedge \ \varphi[a] \tag{8.14}$$

When the base sort S is finite one can use wS1S (weak S1S, where set variables range over finite sets) to encode queue operations `revcons`, `cons`, ϵ, prefix, and suffix [KMS98]. A direct encoding automatically gives the ability to quantify over queues, as well as accessing elements from queues by their index. On the other hand, concatenation of queues cannot be encoded when these are non-lossy and of unbounded length as the results reviewed in the next Section imply. A direct encoding of queue constraints into wS1S also does not support subsequence relations.

8.2.2 Queues as a sub-theory of concatenation

Instead of taking `cons` and `revcons` as primitive queue constructors one could alternatively base a theory of queues on concatenation and formation of singleton queues, and define `cons` and `revcons` as derived operations: $\texttt{cons}(a, q) = [a] \circ q$, $\texttt{revcons}(q, a) = q \circ [a]$. Solving equalities over sequences with concatenation is known as the *word unification problem*. Special cases of the word unification problem were addressed in [Hme76]. Makanin [Mak77] gives an algorithm for word unification showing that word unification problem is decidable. Jaffar [Jaf90] provides a modification of Makanin's algorithm for generating all minimial word unifiers. He notes that infinitely many minimal unifiers may exist. An example is the word equation $ax = xa$, which has unifiers a^*. This equation is also a legal constraint between queues, and we show how to represent the infinitely many unifiers with one constraint.

Our solver based approach in the integration of decision procedures is precisely limited to theories where the set of possibly implied equalities can be represented by a finite quotient. gives a more streamlined and bug-free presentation with generalizations. Although 3-SAT can be immediately reduced to word unification with a one-character alphabet, showing that word unification is NP-hard, the general word unification problem has caused more pains to implement efficiently. Makanin's algorithm requires at most doubly exponential time [Gut98]. Büchi and Senger [BS86] show that the word disunification problem is reducible to the word unification problem. I.e., given a disequation $v \neq w$ we can effectively (and very simply) construct words v' and w' such that

$$(\exists * \ . \ v \neq w) \quad \equiv \quad (\exists * \ . \ v' = w')$$

The full first-order equational theory of words is unfortunately undecidable as shown by Quine [Qui46]. The paper gives a number of constructions in this end. The first and simplest uses concatenation for addition. Multiplication is encoded using a string-based encoding of finite relations. To encode that $x \times y = z$ he encodes a finite relation of pairs consisting of finite relations. To encode that $x \times y = z$ he encodes a finite relation of pairs consisting of

$$(\underbrace{a}_{x}, \underbrace{a}_{y}), (\underbrace{a}_{x-1}, \underbrace{a}_{2y}), \ldots, (\underbrace{a}_{1}, \underbrace{a}_{x \times y})$$

The entire finite binary relation can be encoded in a single string w by segments of the form $bzbubzbvbzb$ where (u, v) is a pair in the relation and z is a string of a's longer than any of the u's and v's. Intuitively, this can be achieved by requireing the existence of substring $bzbubzbvbzb$ in w, such that any substring of pure a's in w is a substring of z. Notice that the encoding uses concatentation of strings in an essential way.

We do not at present know of a way to reduce negated subsequence relations to the existential fragment of word equations.

8.3 A decision procedure for queues

To present the ground decision procedure for queues we use concatenation as primitive constructor rather than `cons` and `revcons`. Queues are then simply sequences with a at most one non-atomic component. In fact all transformations by the solver are sound for sequences that contain more than one non-atomic component.

We write queue terms succinctly as strings with the following conventions: a, b range over individual atoms; A, B range over possibly empty strings of atoms; x, y range over queue variables; v, w range over arbitrary queues (i.e., are of the form A or AxB); let $A = a_1 a_2 \ldots a_k$ then $A[i : j]$ denotes the sequence $a_l a_{l+1} \ldots a_u$, when $l = \max(1, i) \leq \min(k, j) = u$, and when $\max(1, i) > \min(k, j)$, then $A[i : j] = \epsilon$. If v is a queue, then $|v|$ is the length of v. To reverse a sequence of atoms A we write $(A)^R$.

8.3.1 Selectors

The canonizer σ is used to handle selectors. Given a constraint over queue expressions the canonizer σ tries where possible to evaluate selectors when applied to queues. If a selector is applied to a term where it is not possible to immediately evaluate the selector we accumulate in $\mathcal{C}$ disjunctions of new constraints for later splitting. In the end all selectors have been eliminated or are applied only to terms of the form:

$$\texttt{head}(\epsilon), \quad \texttt{tail}(\epsilon), \quad \texttt{first}(\epsilon), \quad \texttt{last}(\epsilon) \ .$$

The rest of the solver treats terms whose main connective is a selector as uninterpreted. The effect of canonization on selectors is summarized in Figure 8.4, where the immediate arguments of the selectors are assumed canonized.

$\sigma(\mathcal{C}, \texttt{head}([a] \circ q_2))$	$=$	$(\mathcal{C}, a)$
$\sigma(\mathcal{C}, \texttt{head}(q_1 \circ [a] \circ q_2))$	$=$	$(\mathcal{C} \cup \{q_1 \doteq \epsilon \ \vee \ q_1 \doteq [b] \circ y\}, \texttt{head}(q_1 \circ [a]))$
$\sigma(\mathcal{C}, \texttt{head}(q_1))$	$=$	$(\mathcal{C} \cup \{q_1 \doteq \epsilon \ \vee \ q_1 \doteq [b] \circ y\}, \texttt{head}(q_1))$
$\sigma(\mathcal{C}, \texttt{head}(\epsilon))$	$=$	$(\mathcal{C}, \texttt{head}(\epsilon))$
$\sigma(\mathcal{C}, \texttt{tail}([a] \circ q_2))$	$=$	$(\mathcal{C}, q_2)$
$\sigma(\mathcal{C}, \texttt{tail}(q_1 \circ [a] \circ q_2))$	$=$	$(\mathcal{C} \cup \{q_1 \doteq \epsilon \ \vee \ q_1 \doteq [b] \circ y\}, \texttt{tail}(q_1 \circ [a]) \circ q_2)$
$\sigma(\mathcal{C}, \texttt{tail}(x \circ q_1))$	$=$	$(\mathcal{C} \cup \{x \doteq \epsilon \ \vee \ x \doteq [b] \circ y\}, \texttt{tail}(x \circ q_1))$
$\sigma(\mathcal{C}, \texttt{tail}(\epsilon))$	$=$	$(\mathcal{C}, \texttt{tail}(\epsilon))$

Figure 8.4: Canonization of selectors

In the figure, q and q_1 are of the form $x_1 \circ x_2 \circ \ldots \circ x_n$ i.e., concatenations of queue variables, and q_2 is an arbitrary concatenation of queue expressions, i.e., of the form $x_1 \circ [a_1] \circ \cdots \circ x_n \circ [a_m]$. The atom b and queue variable y are fresh. Canonization rules for dual operators `first` and `last` are similar to the rules for `head` and `tail` and are therefore not listed.

8.3.2 Equations

Equations are solved in three stages. Assume that the constraint context $\mathcal{C}$ is of the form $\mathcal{E} \wedge D$, where $\mathcal{E}$ is a set (interpreted as a conjunction) of residue equalities (unsolved equalities), and $\mathcal{D}$ consists of disjunctions or other non-equational constraints. When adding a new equality constraint $v \doteq w$, as well as any other constraint, we set the default effect of $addConstraint(\mathcal{C}, c)$ to $(\mathcal{C} \cup \{c\}, [\,])$, and then normalize the residue equalities $\mathcal{E}$ relative to the new constraint to extract a substitution and updated residue equalities. Invocations of

split finally eliminates all equalities in $\mathcal{E}$ producing a context $\mathcal{C}$ without residue equalities.

Normalizing residue equalities: In augmenting $\mathcal{C} : \mathcal{E} \cup \mathcal{D}$ with an equality constraint $v \doteq w$ we apply the transformations in Figure 8.5 with the initial constraint

$$(\mathcal{E} \cup \{v \doteq w\}, [\,]) \ .$$

to produce either FAIL or the pair $(\mathcal{E}', \theta')$. In the first case we set the effect of $addConstraint(\mathcal{C}, v \doteq w)$ to $(\texttt{false}, [\,])$. In the other case we set $addConstraint\ (\mathcal{E} \cup \mathcal{D}, v \doteq w)$ to $((\mathcal{E}' \cup \mathcal{D})\theta', \theta')$.

(fail)	$(\mathcal{E} \cup \{v \doteq w\}, \theta)$	$\mapsto$	FAIL if v is a proper subterm of w
(decompose)	$(\mathcal{E} \cup \{av \doteq bw\}, \theta)$	$\mapsto$	$(\mathcal{E} \cup \{v \doteq w\}, \theta)[a \mapsto b]$
(decompose)	$(\mathcal{E} \cup \{va \doteq wb\}, \theta)$	$\mapsto$	$(\mathcal{E} \cup \{v \doteq w\}, \theta)[a \mapsto b]$
(simplify)	$(\mathcal{E} \cup \{v \doteq v\}, \theta)$	$\mapsto$	$(\mathcal{E}, \theta)$
(reduce)	$(\mathcal{E} \cup \{x \doteq w\}, \theta)$	$\mapsto$	$(\mathcal{E}, \theta)[x \mapsto w]$

Figure 8.5: Rules for decomposing equalities

Rules (fail), (decompose), (simplify), and (reduce) are applied in decreasing order of preference. When no rules from Figure 8.5 apply to $\mathcal{E}$, each remaining equality takes the form

$$Ax \ \doteq \ yB$$

where A and B are non-empty sequences of atoms.

Elimination of connecting residues: A context $\mathcal{C}$ containing the constraint $Ax \doteq yB$, where $|A| \leq |B|$ and x and y are different queue variables, can be simplified using the following application of *split*:

$$\begin{aligned} split(\mathcal{C}) \quad = \quad & \langle addConstraint(\mathcal{C}, y \doteq A[1:j]) \mid j = 0 \ldots |A| - 1 \rangle, \\ & \langle addConstraint(\mathcal{C}, x \doteq zB) \rangle \qquad \text{where } z \text{ is fresh} \end{aligned} \tag{8.15}$$

By maintaining substitutions in a triangular form each descendent requires the same or less space as the parent as lemma 8.3.1 shows:

Lemma 8.3.1 (Complexity) *Let*

$$M_{\mathcal{E}} = \sum_{x \in Vars(\mathcal{E})} \max \{|A|, |B| \mid AxB \in \mathcal{E}\}$$

and let $(\mathcal{E}', \theta')$ *be a branch obtained by eliminating a variable in* $\mathcal{E}$*, then*

$$M_{\mathcal{E}'} \leq M_{\mathcal{E}} \ .$$

Proof:

Given any pair $(\mathcal{E}, \theta)$, let

$$M = \sum_{x \in Vars(\mathcal{E})} \max\{|A|, |B| \mid AxB \in \mathcal{E}\}$$

Then

1. Each simplification step in Figure (8.5) does not increase M. Trivially (decompose) can not increase M. In the (reduce) step, assume that we are applying the substitution $x \mapsto AyB$, where

$$\begin{array}{rcl} k & = & \max\{|A|, |B| \mid AxB \in \mathcal{E}\} \\ l & = & \max\{|A|, |B| \mid AyB \in \mathcal{E}\} \end{array}$$

   then M' corresponding to $(\mathcal{E}, \theta)[x \mapsto AyB]$ satisfies

$$\begin{array}{rcl} M' & = & M - k - l + \\ & & \max(\{|A'A|, |BB'| \mid A'xB' \in \mathcal{E}\} \cup \{|A|, |B| \mid AyB \in \mathcal{E}\}) \\ & \leq & M - k - l + \max(\max(|A|, |B|) + k, l) \\ & \leq & M \end{array}$$

2. Variable elimination does not increase M. In a variable elimination step we apply the substitutions $x \mapsto zB, y \mapsto Az$ corresponding to the constraint $Ax \doteq yB$. Then

$$\begin{array}{rcccl} |A| & \leq & k & = & \max\{|A|, |B| \mid AxB \in \mathcal{E}\} \\ |B| & \leq & l & = & \max\{|A|, |B| \mid AyB \in \mathcal{E}\} \end{array}$$

   and

$$\begin{array}{rcl} M' & = & M - k - l + \max(\{|CA|, |D| \mid CyD \in \mathcal{E}\} \cup \{|C|, |BD| \mid CxD \in \mathcal{E}\}) \\ & \leq & M - k - l + \max(l + |A|, k + |B|) \\ & \leq & M \end{array}$$

∎

Elimination of looping residues: The first two transformations on $\mathcal{C}$ leave us with residues of the form $Ax \doteq xB$, where A, B can be assumed to be non-empty sequences of atoms of the same length. If A and B are not of the same length then $\mathcal{C}$ is unsatisfiable and replaced by `false`. We eliminate residues of this form using *split* with the effect:

$$\begin{array}{l} split(\{Ax \doteq xB\} \cup \mathcal{C}) = \\ \quad \langle addConstraints(\mathcal{C}, \{x \doteq A[1:j], B = wrap(A, j)\} \mid j = 0 \ldots |A| - 1\rangle, \qquad (8.16) \\ \quad \langle addConstraints(\mathcal{C}, \{\mathsf{periodic}(x, j, A, |A|), B = wrap(A, j)\}) \mid j = 0 \ldots |A| - 1\rangle \end{array}$$

where we use a new predicate periodic. It is treated as a primitive relation, but we intend the interpretation

$$\mathsf{periodic}(x, j, A, l): \quad l \le |x| \quad \wedge \quad |x| \equiv j \pmod{|A|} \quad \wedge \quad x = ext(A, |x|)$$

where

$$wrap(A, j) \stackrel{\text{def}}{=} \textbf{if } j = 0 \textbf{ then } A \textbf{ else } A[j+1 : |A|]A[1 : j] \tag{8.17}$$

$$ext(A, l) \stackrel{\text{def}}{=} \textbf{if } |A| \le l \textbf{ then } A[1 : l] \textbf{ else } Aext(A, l - |A|) \tag{8.18}$$

Combinations of periodic**:** We can maintain at most one occurrence of $\mathsf{periodic}(x, j, A, l)$ for every x by supplying the splitting rule:

$$\begin{aligned} &split(\{\mathsf{periodic}(x, j, A, l), \mathsf{periodic}(x, k, B, m)\} \cup \mathcal{C}) = \\ &\left\langle addConstraints(\mathcal{C}, \left\{ \begin{array}{l} x \doteq ext(A, i), \\ x \doteq ext(B, i) \end{array} \right\}) \;\middle|\; \begin{array}{l} i = \max(l, m) \ldots n-1, \\ i \equiv j \pmod{|A|} \\ \equiv k \pmod{|B|} \end{array} \right\rangle \\ &\langle addConstraints(\mathcal{C}, \left\{ \begin{array}{l} \mathsf{periodic}(x, k, A[1 : g], n), \\ A \doteq ext(A[1 : g], |A|), \\ B \doteq ext(B[1 : g], |B|) \end{array} \right\}) \mid j \equiv k \pmod{g} \rangle \end{aligned} \tag{8.19}$$

where

$$g \stackrel{\text{def}}{=} \gcd(|A|, |B|) \qquad n \stackrel{\text{def}}{=} |A| + |B| - g \ .$$

We see that the first branches cover the cases where $|x|$ ranges from $\max(l, m)$ to $n-1$. The cases where $|x| \ge n$ have been collapsed into a single branch. To establish that this preserves soundness we use a lemma which has also been useful in [BP98]:

Lemma 8.3.2 *With A, B, n, and g as above, and $m \ge n$, then*

$$ext(A, m) = ext(B, m) \quad \leftrightarrow \quad ext(A, n) = ext(B, n)$$

Thus, unfolding A and B beyond n does not introduce any new constraints. The extended Chinese Remainder Theorem is used to combine the length constraints on x.

Factorization of periodic**:** When a substitution replaces a variable x by a compound term we normalize periodic using transformations

$$\begin{array}{lcl} \mathsf{periodic}(\epsilon, i, A, n) & \mapsto & i = 0 \ \wedge \ n < 0 \\ \mathsf{periodic}(wa, i, A, n) & \mapsto & a \doteq A[i : i] \wedge \mathsf{periodic}(w, i-1 \bmod |A|, A, n-1) \\ \mathsf{periodic}(aw, i, A, n) & \mapsto & a \doteq A[1 : 1] \ \wedge \ \mathsf{periodic}(w, i-1 \bmod |A|, wrap(A, 1), n-1) \end{array}$$

Context dependent canonization: Since the predicate periodic has been introduced to summarize constraints of the form $Ax \doteq xB$ we need to instrument the canonizer σ with

this information in order to canonize Ax and xB to the same term. This can be achieved by shifting queue variables to the left as much as possible:

$$\sigma(\mathcal{C}, Bx) \quad = \quad (\mathcal{C}, xext(wrap(A, j), |B|)) \qquad (8.20)$$
$$\text{if } (B)^R = ext((A)^R, |B|) \text{ and } \mathsf{periodic}(x, i, A, l) \in \mathcal{C} \quad .$$

Theorem 8.3.8 implies that the resulting effect of σ derives all implied equalities. In particular a disequality $q \not\equiv r$ is inconsistent if and only if q and r canonize to the same terms once all constraints in $\mathcal{C}$ have been processed.

8.3.3 Subsequences

Having solved equational constraints we are ready to solve disequational constraints. These constraints are solved using applications of *split*, which transforms disequational constraints into normal form, from which an injective model can be extracted.

Uncontextual simplifications: The most basic such transformations are given in Figure 8.6 and for negated constraints in Figure 8.7. We have used the shorthand

$$c \mapsto c_1 \ \vee \ c_2 \wedge c_3$$

to encode that

$$split(\{c\} \cup \mathcal{C}) = \langle addConstraint(\mathcal{C}, c_1), \ addConstraints(\mathcal{C}, \{c_2, c_3\})\rangle \ .$$

The rules for suffix follow a similar pattern as for prefix.

These transformations turn an occurrence of $v \preceq w$ into a disjunction of conjunctions, where each conjunct is either an equality constraint or of the form

$$\begin{array}{lll} AxB \preceq y, & \mathsf{prefix}(xA, y), & \mathsf{suffix}(Ax, y), \\ AxB \not\preceq y, & \neg\mathsf{prefix}(xA, y), & \neg\mathsf{suffix}(Ax, y) \end{array}$$

The combined effect so far can be summarized as

Lemma 8.3.3 *Let $\mathcal{C}$ be a conjunction of literals such that none of the rules in Figures 8.6, 8.7, 8.5, apply, then the literals in $\mathcal{C}$ are of the form:*

$$\begin{array}{llll} AxB \preceq y, & \mathsf{prefix}(xA, y), & \mathsf{suffix}(Ax, y), & v \not\equiv w, \\ AxB \not\preceq y, & \neg\mathsf{prefix}(xA, y), & \neg\mathsf{suffix}(Ax, y), & \mathsf{periodic}(x, i, C, l) \ . \end{array}$$

Contextual simplifications: A contextual transformation rule depends on at least two constraints in $\mathcal{C}$ and simplify the set of constraints. For instance, if $\mathcal{C}$ contains $\mathsf{prefix}(u, w)$ and $\mathsf{prefix}(v, w)$ for two different u and v, then as prefix is a linear order, we can simplify $\mathcal{C}$ by replacing these constraints with either

$$\mathsf{prefix}(u, v) \ \wedge \ \mathsf{prefix}(v, w) \quad \text{or} \quad \mathsf{prefix}(v, u) \ \wedge \ \mathsf{prefix}(u, w) \ .$$

$$
\begin{array}{llll}
v \preceq aw & \mapsto & \mathsf{prefix}(v, aw) \ \vee \ v \preceq w & w \preceq w \ \mapsto \ \mathtt{true} \\
v \preceq wa & \mapsto & \mathsf{suffix}(v, wa) \ \vee \ v \preceq w & \epsilon \preceq w \ \mapsto \ \mathtt{true} \\
w \preceq \epsilon & \mapsto & w \doteq \epsilon & \mathsf{prefix}(v, v) \ \mapsto \ \mathtt{true} \\
\mathsf{prefix}(v, wb) & \mapsto & v \doteq wb \ \vee \ \mathsf{prefix}(v, w) & \mathsf{prefix}(v, \epsilon) \ \mapsto \ \epsilon \doteq v \\
\mathsf{prefix}(av, bw) & \mapsto & a \doteq b \wedge \mathsf{prefix}(v, w) & \mathsf{prefix}(\epsilon, v) \ \mapsto \ \mathtt{true}
\end{array}
$$

$$
\begin{array}{lll}
\mathsf{prefix}(xA, bw) \ \mapsto \ \left(\begin{array}{l} x \doteq \epsilon \wedge \mathsf{prefix}(A, bw) \\ \vee \ \ x \doteq by \ \wedge \ \mathsf{prefix}(yA, w) \end{array} \right) & x \notin w, y \text{ is fresh} \\
\mathsf{prefix}(av, x) \ \mapsto \ x \doteq ay \ \wedge \ \mathsf{prefix}(v, y) & x \notin v, y \text{ is fresh} \\
\mathsf{prefix}(Ax, x) \ \mapsto \ |A| = 0 &
\end{array}
$$

$$
\mathsf{prefix}(xA, Bx) \ \mapsto \ |A| \leq |B| \ \wedge \ \bigvee_{j=0}^{|B|-1} Bx \doteq xAext(wrap(B, j), |B| - |A|)
$$

Figure 8.6: Uncontextual positive simplifications

We refer to Figure 8.8 for the complete set of contextual transformation rules.

Rules for periodic suffix and suffix^2 are similar to periodic prefix respectively prefix^2 and have not been included in Figure 8.8.

An effect of the contextual transformation rules is that they ensure that prefix and suffix are linear orders and $\preceq$ is a partial order, which we describe below.

Definition 8.3.4 (Partial variable ordering $\sqsubset$) *Let $\mathcal{C}$ be a conjunction of constraints, then $\sqsubset$ is the transitive closure of the binary relation defined by*

$$
x \sqsubset y \quad \stackrel{\text{def}}{=} \quad \mathsf{prefix}(xA, y) \in \mathcal{C}, \ \mathsf{suffix}(Ax, y) \in \mathcal{C}, \ \textit{or} \ AxB \preceq y \in \mathcal{C}
$$

We now have:

Lemma 8.3.5 *Let $\mathcal{C}$ be a conjunction of constraints closed under equality, disequality, uncontextual simplifications and the contextual simplifications from Figure 8.8, then*

- *The relation $\sqsubset$ is a partial ordering of the queue variables.*
- *For every queue variable y there is at most one constraint $\mathsf{prefix}(w, y)$ and at most one constraint $\mathsf{suffix}(w, y)$ in $\mathcal{C}$.*
- *If $\mathcal{C}$ contains a constraint $\mathsf{periodic}(x, j, A, l)$, then there are no constraints of the form $\neg\mathsf{prefix}(w, x)$, $\neg\mathsf{suffix}(w, x)$, or $w \not\preceq x$ in $\mathcal{C}$.*

$$
\begin{array}{lcll}
v \not\preceq aw & \mapsto & \neg\mathsf{prefix}(v, aw) \ \wedge \ v \not\preceq w & \\
v \not\preceq wa & \mapsto & \neg\mathsf{suffix}(v, wa) \ \wedge \ v \not\preceq w & \\
\epsilon \not\preceq w & \mapsto & \texttt{false} & \\
w \not\preceq \epsilon & \mapsto & w \neq \epsilon & \\
w \not\preceq w & \mapsto & \texttt{false} & \\
\neg\mathsf{prefix}(av, bw) & \mapsto & a \neq b \ \vee \ \neg\mathsf{prefix}(v, w) & \\
\neg\mathsf{prefix}(v, wb) & \mapsto & v \neq wb \ \wedge \ \neg\mathsf{prefix}(v, w) & \\
\neg\mathsf{prefix}(xA, bw) & \mapsto & \left(\begin{array}{ll} & x \doteq \epsilon \ \wedge \ \neg\mathsf{prefix}(A, bw) \\ \vee & \exists y, c \ . \ x \doteq cy \ \wedge \ \neg\mathsf{prefix}(cyA, bw) \end{array}\right) & x \notin w \\
\neg\mathsf{prefix}(av, x) & \mapsto & \left(\begin{array}{ll} & x \doteq \epsilon \\ \vee & \exists y, b \ . \ x \doteq by \ \wedge \ \neg\mathsf{prefix}(av, by) \end{array}\right) & x \notin v \\
\neg\mathsf{prefix}(v, \epsilon) & \mapsto & \epsilon \neq v & \\
\neg\mathsf{prefix}(\epsilon, v) & \mapsto & \texttt{false} & \\
\neg\mathsf{prefix}(Ax, x) & \mapsto & |A| > 0 & \\
\neg\mathsf{prefix}(xA, Bx) & \mapsto & |A| > |B| \ \vee \ \bigwedge_{j=0}^{|B|-1} xAext(wrap(B, j), |B| - |A|) \neq Bx &
\end{array}
$$

Figure 8.7: Uncontextual negative simplifications

Saturation rules: Using $c_1 \wedge c_2 \hookrightarrow c_3$ as shorthand for $c_1 \wedge c_2 \mapsto c_1 \wedge c_2 \wedge c_3$ we finally saturate $\mathcal{C}$ with rules such as

$$\mathsf{periodic}(y, i, C, l) \ \wedge \ \mathsf{prefix}(xA, y) \hookrightarrow \bigvee_{j=0}^{|C|-1} xAwrap(C, j) \doteq CxA$$

to guarantee that the following lemma holds

Lemma 8.3.6

- *If* $\mathsf{periodic}(y, i, C) \in \mathcal{C}$ *and* $w \preceq y \in \mathcal{C}$*, then* w *is constrained by* $\mathsf{periodic}(w, j, wrap(C, k))$ *in* $\mathcal{C}$ *for some* j *and* k*. Similar statements hold for* $\mathsf{prefix}(w, y)$ *and* $\mathsf{suffix}(w, y)$*.*
- *If* $w \not\preceq y$*, then for every subsequence* v *of* y $\mathcal{C}$ *implies that* w *is not a subsequence of* v*.*

The full set of saturation rules required for Lemma 8.3.6 is given in Figure 8.9.

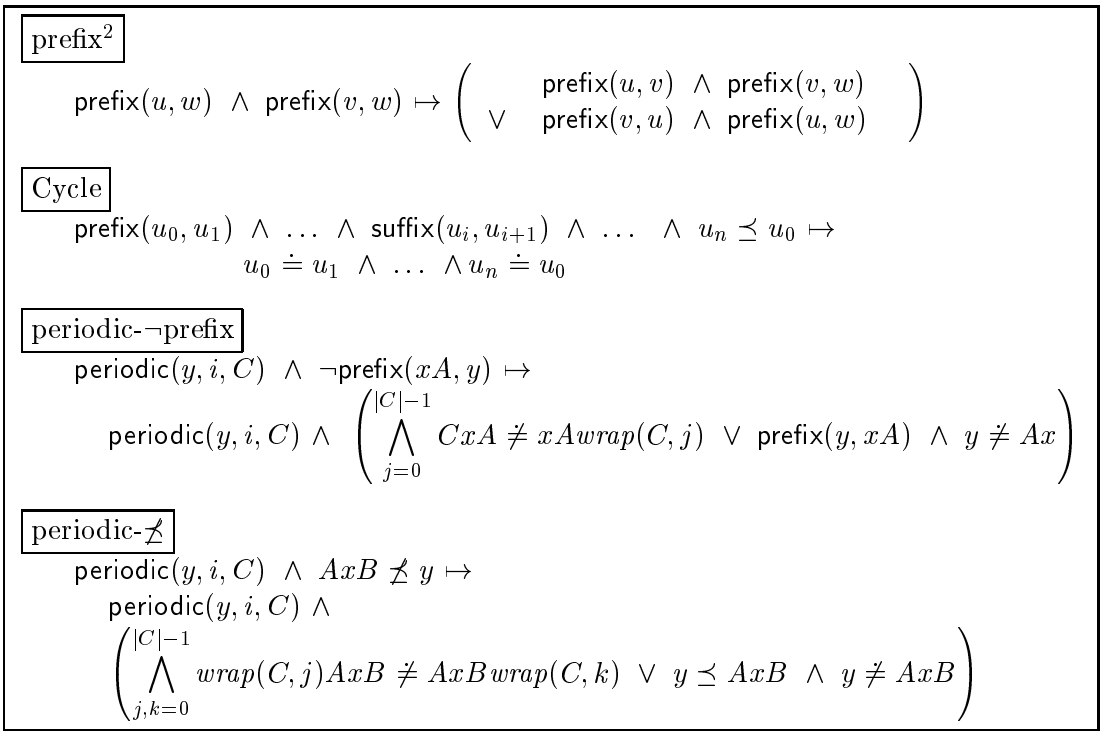


Figure 8.8: Contextual simplifications

8.3.4 Correctness, Complexity and Completeness

A simple inspection reveals:

Theorem 8.3.7 (Soundness) *All rules preserve satisfiability.*

The accumulated effect of the splits works both as a satisfiability checker and generator of an injective model, which is important for obtaining a complete integration.

Theorem 8.3.8 (Completeness) *If a context $\mathcal{C}$ is closed under all splitting rules, then*

1. *$\mathcal{C}$ is satisfiable.*
2. *For any terms q_i and r_i, $\mathcal{C} \models \bigvee_i q_i = r_i$ iff $\sigma_{\mathcal{C}}(q_i) = \sigma_{\mathcal{C}}(r_i)$ for some i.*

Proof:

Outline From a context $\mathcal{C}$ closed under all splitting rules we construct an injective model by differentiating all atoms that have not been eliminated, and starting with the smallest elements in the partial order $\sqsubset$ we build different realizations for the queue variables.

periodic prefix

$$\mathsf{periodic}(y,i,C,l) \ \wedge \ \mathsf{prefix}(xA,y) \hookrightarrow \bigvee_{j=0}^{|C|-1} xA\,wrap(C,j) \doteq CxA$$

periodic-$\preceq$

$$\mathsf{periodic}(y,i,C,l) \ \wedge \ AxB \preceq y \hookrightarrow \bigvee_{j,k=0}^{|C|-1} AxB\,wrap(C,j) \doteq wrap(C,k)AxB$$

¬prefix-prefix	$\neg\mathsf{prefix}(v,y) \wedge \mathsf{prefix}(w,y)$	$\hookrightarrow$	$\neg\mathsf{prefix}(v,w)$
¬suffix-suffix	$\neg\mathsf{suffix}(v,y) \wedge \mathsf{suffix}(w,y)$	$\hookrightarrow$	$\neg\mathsf{suffix}(v,w)$
$\not\preceq$-$\preceq$	$v \not\preceq y \ \wedge \ w \preceq y$	$\hookrightarrow$	$v \not\preceq w$
$\not\preceq$-prefix	$v \not\preceq y \ \wedge \ \mathsf{prefix}(w,y)$	$\hookrightarrow$	$v \not\preceq w$
$\not\preceq$-suffix	$v \not\preceq y \ \wedge \ \mathsf{suffix}(w,y)$	$\hookrightarrow$	$v \not\preceq w$

Figure 8.9: Saturation rules

Initially let $\eta_0 : \mathcal{V} \ \mapsto \ \Sigma^*$ be a map with empty domain. A full evaluation of all queue variables is extracted in stages starting from a queue variable that has no sub-queues. Let η_n be the partial evaluation of queue variables extracted at stage n, and y be the n'th queue variable to be processed. We distinguish two cases:

Periodic $\mathsf{periodic}(y,i,C) \in \mathcal{C}$ for some i and C. Then whenever $w \preceq y \in \mathcal{C}$, then by Lemma 8.3.6 $\mathcal{C}$ implies $\mathsf{periodic}(w,j,wrap(C,k))$ for some j and k.

Also by Lemma 8.3.5 there are no constraints $AxB \not\preceq y$, $\neg\mathsf{prefix}(xA,y)$ or $\neg\mathsf{suffix}(Ax,y)$.

Let

$$m \ = \ \max\left(\begin{array}{ll} & \{|\eta_n(CzD)| \mid CzD \not\equiv AyB \in \mathcal{C}, z \ll y\} \\ \cup & \{|\eta_n(w)| \mid w \preceq y, \mathsf{prefix}(w,y), \quad \text{or } \mathsf{suffix}(w,y) \in \mathcal{C}\} \end{array}\right)$$

and set

$$\eta_{n+1} \ := \ \eta_n \ \cup \ [y \mapsto ext(C, m|C| + i)]$$

It is then straight-forward to verify that η_{n+1} satisfies all disequalities and sub-queue relations between queue expressions whose variables are in the domain of η_{n+1}.

Aperiodic If it is not the case that y is constrained by $\mathsf{periodic}(y,i,C)$, then assume

we have the following constraints on y:

$$\mathsf{prefix}(x_0A_0, y), \quad A_ix_iB_i \preceq y, \quad \text{for } i = 1, \ldots, k, \quad \mathsf{suffix}(A_{k+1}x_{k+1}, y)$$

Set

$$\eta_{n+1} \;:=\; \eta_n \;\cup\; [y \mapsto x_0A_0z_1A_1x_1B_1 \ldots z_kA_kx_kB_kz_{k+1}A_{k+1}x_{k+1}],$$

where z_i are queues of length at least one containing fresh atoms such that $|y|$ is longer than $\max\{|CzD| \mid CzD \neq AyB \in \mathcal{C}, z \ll y\}$.
The saturation rules ensure that whenever $AxB \not\preceq y$, $\neg\mathsf{prefix}(xA, y)$ or $\neg\mathsf{suffix}(Ax, y)$ is asserted, then we may assume by induction on n that that these are not sub-queues of the sub-queues of y. The use of fresh atoms in the z_i prevents any of these queues to be sub-queues of any other part of y.

∎

Theorem 8.3.9 (Complexity) *The satisfiability problem for constraints over queues is NP-complete, and our procedure is in NP.*

Proof outline:

By inspecting the constraint solving steps we see that each branch can be represented in space bounded by the size of the input. The disjunctive splitting causes branches of at most polynomial depth. The theory is on the other hand NP-hard. We can reduce an arbitrary instance $(\mathcal{V}, \mathcal{E})$ of the graph 3 coloring problem to the constraints:

$$\mathtt{r} \neq \mathtt{g} \;\wedge\; \mathtt{g} \neq \mathtt{b} \;\wedge\; \mathtt{b} \neq \mathtt{r} \;\;\wedge \bigwedge_{\mathtt{v} \in \mathcal{V}} \mathtt{v}\mathtt{v}'\mathtt{v}''x_v = x_v\mathtt{rgb} \;\wedge \bigwedge_{(\mathtt{v},\mathtt{w}) \in \mathcal{E}} \mathtt{v} \neq \mathtt{w}$$

∎

8.4 Implementation

The present prototype implementation of the decision procedures for queues uses concatenation as the basic constructor. Consequently constraints of the form

$$x \circ [a] \circ y \doteq [b] \circ x \circ z$$

are legal inputs and are decomposed to

$$\{x \circ [a] \doteq [b] \circ x,\; y \doteq z\} \;.$$

Presently the constraints involving the predicate periodic are not generated.

8.4.1 Arithmetical integration

Similarly to recursive data-types one can add a length function on queues. The empty queue is given length zero, and every application of `revcons` and `cons` contributes by incrementing the length by one. This is summarized by the effect of canonization in Figure 8.4.

Unfortunately we do not have a complete integration of arithmetic and sub-queue relations. Instead we use an incomplete combination with the arithmetic solver via `SUP` and `INF` to access lower and upper bounds on variables. If a variable has a positive lower bound it is replaced by a fresh instance of the length of the bound. If the length of a queue variable has a finite upper bound we enumerate the possible instances.

Besides being essential in establishing the example from Section 8.1, support for the length function was noted essential in small lemmas from [NG98]. They are established automatically here using the decision procedures. In general one should note that the automatic support allows to avoid having to state and prove such lemmas separately.

$\texttt{macro cadr}(l) = \texttt{head}(\texttt{tail}(l))$
$\texttt{macro caddr}(l) = \texttt{head}(\texttt{tail}(\texttt{tail}(l)))$

$\|l\| \geq 1 \ \rightarrow l = \texttt{cons}(\texttt{head}(l), \texttt{tail}(l))$	0.01
$\texttt{cons}(m_1, l_1) = \texttt{cons}(m_2, l_2) \ \rightarrow \ m_1 = m_2 \wedge l_1 = l_2$	0.00
$\|l\| = 1 \ \rightarrow l = [\texttt{head}(l)]$	0.01
$\|l\| = 2 \ \rightarrow l = [\texttt{head}(l), \texttt{cadr}(l)]$	0.02
$\|l\| = 3 \ \rightarrow l = [\texttt{head}(l), \texttt{cadr}(l), \texttt{caddr}(l)]$	0.03
$\|l_1 \circ l_2\| = \|l_1\| + \|l_2\|$	0.00
$\|l_1\| = 2 \wedge \|l_2\| \geq 1 \ \rightarrow \ \texttt{caddr}(l_1 \circ l_2) = \texttt{head}(l_2)$	0.05
$\|l_1\| \geq 2 \ \rightarrow \ \texttt{cadr}(l_1 \circ l_2) = \texttt{cadr}(l_1)$	0.03
$l_2 = \texttt{cons}(m, l_1) \ \rightarrow \|l_2\| = \|l_1\| + 1$	0.00
$\texttt{cons}(m, l_1) = l_2 \ \rightarrow \ \|l_2\| \geq 2 \ \rightarrow$ $\texttt{cadr}(l_2) = \texttt{head}(l_1) \ \wedge \ \texttt{tail}(l_2) = l_1$	0.01
$\|l_2\| \geq 1 \vee \|l_1\| \geq 1 \ \rightarrow \ \texttt{head}(l_1 \circ l_2) =$ **if** $l_1 = \texttt{empty}$ **then** $\texttt{head}(l_2)$ **else** $\texttt{head}(l_1)$	0.06

Figure 8.10: Lemmas from [NG98]

8.4.2 Other examples

The buffer system discussed in [Sha93] provided some of the early motivation for developing decision support for queues. The decision procedures developed in this chapter trivially

establishes all verification conditions associated with this example, including:

Initially $\epsilon = \epsilon \circ \epsilon \circ \overline{\bot}$

`read` $input\_h = output\_h \circ b \circ \overline{input} \ \wedge \ input \neq \bot \ \rightarrow$
$input\_h = output\_h \circ \texttt{revcons}(b, input) \circ \overline{\bot}$

`write` $input\_h = output\_h \circ b \circ \overline{input} \wedge b \neq \epsilon \ \rightarrow$
$input\_h = \texttt{revcons}(output\_h, \texttt{head}(b)) \circ \texttt{tail}(b) \circ \overline{input}$

where

$$\overline{x} \ \stackrel{\text{def}}{=} \ \textbf{if} \ \ x = \bot \ \textbf{then} \ \ \epsilon \ \textbf{else} \ [x]$$

which need not be established using induction, but directly using the decision procedures.

In [Fis98] a software retrieval system for functions manipulating lists is discussed. Bernd Fischer kindly provided around 15,000 formulas including the relations $\preceq$, prefix(,), and suffix(,); and functions `head`, `tail`, `cons`, and $\circ$. A large fragment of the formulas included also predicates for ordered lists. These predicates were left uninterpreted in our tests. Our implementation of the decision procedures given in this Chapter together with the quantifier instantiation heuristics was able to automatically establish 1,266 of the 1,800 valid formulas, while spending in average 0.20 seconds on each formula, valid or not. Of the remaining verification conditions it was possible to identify only four valid formulas that were in the scope of the decision procedures, but where quantifier instantation had failed to properly find the right instantiations. In contast a good resolution theorem prover (SPASS, Gandalf, or SETHEO) requires about ten seconds to prove as many verification conditions given appropriate sets of axioms to work with. With a time limit of 90 seconds, however, SPASS and Gandalf outperform our implementation proving up to 1,500 of the 1,800 conditions. One can therefore be tempted to conclude that even simple decision procedures offer competitive performance for the common case to well-tuned general theorem provers as they tend to provide well-directed pruning of the search-space.

8.5 Open problems

Problem 8.5.1 *How expressive is the first-order theory of queues with the sub-queue relation? In particular, how does this compare with the theory of concatenation?*

Problem 8.5.2 *Give a complete decision procedure for a combination of integer linear programming and sub-queue relations.*

Problem 8.5.3 *Extend queue decision procedures with constraints for lists over an ordered domain.*

Problem 8.5.4 *Represent unifiers for the word unification problem using finitely many unifiers as done with bit-vectors.*

8.6 Summary

We gave decision procedures for the universal theory of queues including the sub-queue relationship. Along the way we established that the satisfiability problem for quantifier free formulas with queue constraints is NP-complete.

Bibliography

[Acz88] Peter Aczel. *Non-well-founded sets.* CSLI Lecture Notes, 1988.

[AH96] Rajeev Alur and Thomas A. Henzinger, editors. *Proc. 8^{th} Intl. Conference on Computer Aided Verification*, volume 1102 of *LNCS*. Springer-Verlag, July 1996.

[And81] Peter B. Andrews. Theorem proving via general matings. *J. ACM*, 28(2):193–214, April 1981.

[AS80] Bengt Aspvall and Yossi Shiloach. A fast algorithm for solving systems of linear equations with two variables per equation. *Linear Algebra and its Applications*, 34:117–124, 1980.

[BBM95] Nikolaj S. Bjørner, Anca Browne, and Zohar Manna. Automatic generation of invariants and intermediate assertions. In 1^{st} *Intl. Conf. on Principles and Practice of Constraint Programming*, volume 976 of *LNCS*, pages 589–623. Springer-Verlag, September 1995.

[BBM97] Nikolaj S. Bjørner, Anca Browne, and Zohar Manna. Automatic generation of invariants and intermediate assertions. *Theoretical Computer Science*, 173(1):49–87, February 1997. Preliminary version appeared in 1^{st} *Intl. Conf. on Principles and Practice of Constraint Programming*, vol. 976 of LNCS, pp. 589–623, Springer-Verlag, 1995.

[BC95] Randal E. Bryant and Y.-A. Chen. Verification of arithmetic circuits with binary moment diagrams. In *Proc. of the 32^{nd} ACM/IEEE Design Automation Conference*, 1995.

[BC98] Clark W. Barrett and David Cyrluk. . Private communication, 1998.

[BD94] Jerry R. Burch and David L. Dill. Automatic verification of pipelined microprocessor control. In David L. Dill, editor, *Proc. 6^{th} Intl. Conference on Computer Aided Verification*, volume 818 of *LNCS*, pages 68–80. Springer-Verlag, 1994.

[BDL96] Clark Barrett, David L. Dill, and Jeremy Levitt. Validity checking for combinations of theories with equality. In *1st Intl. Conf. on Formal Methods in Computer-Aided Design*, volume 1166 of *LNCS*, pages 187–201, November 1996.

[BDL98] Clark W. Barrett, David L. Dill, and Jeremy R. Levitt. A decision procedure for bit-vector arithmetic. In *Proc. of the 35^{th} ACM/IEEE Design Automation Conference*, June 1998.

[Bec98] Bernhard Beckert. Rigid E-unification. In Wolfgang Bibel and Peter H. Schmitt, editors, *Automated Deduction — A Basis for Applications*, volume I: Foundations, pages 265–289. Kluwer, Dordrecht, 1998.

[BEFS89] A. Bratsch, H. Eveking, H.-J. Färber, and U. Schellin. LOVERT - A Logic Verifier of Register-Transfer Level Descriptions. In L. Claesen, editor, *IMEC-IFIP International Workshop on Applied Formal Methods for Correct VLSI Design.* Elsevier, 1989.

[BFP92] Peter Baumgartner, Ulrich Furbach, and Uwe Petermann. A unified approach to theory reasoning. Research Report 15-92, Fachbereich Informatik, Universität Koblenz, 1992.

[BG95] Leo Bachmair and Harald Ganzinger. Ordered chaining calculi for first-order theories of binary relations. Technical Report MPI-I-95-2-009, Max Plank Institute, Saarbrücken, 1995. Revised version to appear in JACM.

[Bib82] Wolfgang Bibel. *Automated Theorem Proving.* Friedr. Vieweg & Sohn, Braunschweig, Germany, 1982.

[Bir35] Garrett Birkhoff. On the structure of abstract algebras. In *Cambridge Philosophical Society*, pages 433–454, Trinity College, April 1935.

[Bjø98a] Nikolaj S. Bjørner. Reactive Verification with Queues. In *Engineering Automation for Computer Based Systems*, 1998. To appear, but see `theory.stanford.edu/people/nikolaj`.

[Bjø98b] Nikolaj S. Bjørner. Symbolic temporal tableaux. Draft manuscript, Computer Science Department, Stanford University, 1998. Implemented in STeP.

[BK95] David Basin and Nils Klarlund. Hardware verification using monadic second-order logic. In Pierre Wolper, editor, *Proc. 7^{th} Intl. Conference on Computer Aided Verification*, volume 939 of *LNCS*, pages 31–41. Springer-Verlag, July 1995.

[Ble75] Woody W. Bledsoe. A new method for proving certain Presburger formulas. In *Proc. of the 4^{th} Intl. Joint Conference on Artificial Intelligence*, pages 15–21, September 1975.

[BLM97] Nikolaj S. Bjørner, Uri Lerner, and Zohar Manna. Deductive verification of parameterized fault-tolerant systems: A case study. In *Intl. Conf. on Temporal Logic.* Kluwer, 1997. To appear.

[BLO98] Saddek Bensalem, Yassine Lakhnech, and Sam Owre. Computing abstractions of infinite state systems compositionally and automatically. In Hu and Vardi [HV98], pages 319–331.

[BLS96] Saddek Bensalem, Yassine Lakhnech, and Hassen Saidi. Powerful techniques for the automatic generation of invariants. In Alur and Henzinger [AH96], pages 323–335.

[BM79] Robert S. Boyer and J Strother Moore. *A Computational Logic.* Academic Press, 1979.

[BM88] Robert S. Boyer and J Strother Moore. Integrating decision procedures into heuristic theorem provers: A case study with linear arithmetic. *Machine Intelligence*, 11:83–124, 1988.

[BM96] Jon Barwise and Lawrence Moss. *Vicious Circles.* CSLI publications, 1st edition, 1996.

[BMS95] Anca Browne, Zohar Manna, and Henny B. Sipma. Generalized temporal verification diagrams. In *15th Conference on the Foundations of Software Technology and Theoretical Computer Science*, volume 1026 of *LNCS*, pages 484–498. Springer-Verlag, 1995.

[BMSU97] Nikolaj S. Bjørner, Zohar Manna, Henny B. Sipma, and Tomás E. Uribe. Deductive verification of real-time systems using STeP. In *4th Intl. AMAST Workshop on Real-Time Systems*, volume 1231 of *LNCS*, pages 22–43. Springer-Verlag, May 1997.

[BMSU98] Nikolaj S. Bjørner, Zohar Manna, Henny B. Sipma, and Tomás E. Uribe. Verifying temporal properties of reactive systems: A STeP tutorial. Technical report, Computer Science Department, Stanford University, March 1998.

[BN98] Franz Baader and Tobias Nipkow. *Term Rewriting and all that.* Cambridge University Press, Cambridge, 1998.

[BP98] Nikolaj S. Bjørner and Mark C. Pichora. Deciding fixed and non-fixed size bit-vectors. In *4th Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 1384 of *LNCS*, pages 376–392. Springer-Verlag, 1998.

[Bry86] Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[BS86] J. Richard Büchi and Steven Senger. Coding in the existential theory of concatenation. *Archiv für mathematiche Logik*, 26:101–106, 1986.

[BS93] Franz Baader and Jürgen H. Siekmann. Unification theory. In D.M. Gabbay, C.J. Hogger, and J.A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming.* Oxford University Press, Oxford, UK, 1993.

[BS96] Franz Baader and Klaus U. Schulz, editors. *Frontiers of Combining Systems: Proceedings of the 1st International Workshop.* Kluwer, March 1996.

[BS98] Franz Baader and Klaus U. Schulz. Combination of constraint solvers for free and quasi-free structures. *Theoretical Computer Science*, pages 107–161, February 1998.

[BSU97] Nikolaj S. Bjørner, Mark E. Stickel, and Tomás E. Uribe. A practical integration of first-order reasoning and decision procedures. In *Proc. of the* 14^{th} *Intl. Conference on Automated Deduction*, volume 1249 of *LNCS*, pages 101–115. Springer-Verlag, July 1997.

[BT97] Franz Baader and Cesare Tinelli. A new approach for combining decision procedures for the word problem, and its connection to the Nelson-Oppen combination method. In William McCune, editor, *Proceedings of the 14th International Conference on Automated deduction*, volume 1249 of *LNAI*, pages 19–33, Berlin, July13–17 1997. Springer.

[Buc65] Bruno Buchberger. *Ein Algorithmus zum Auffinden der Basiselemente des Restklassenringes nach einem nulldimensionalen Polynomideal.* Dissertation, Mathematisches Institut der Universität Innsbruck, Innsbruck, Österreich, 1965.

[Bür91] Hans-Jürgen Bürckert. *A Resolution Principle for a Logic with Restricted Quantifiers*, volume 568 of *LNAI.* Springer-Verlag, 1991.

[CC77] Patrick Cousot and Rhadia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In 4^{th} *ACM Symp. Princ. of Prog. Lang.*, pages 238–252. ACM Press, 1977.

[CC90] Domenico Cantone and V. Cutello. A decidable fragment of the elementary theory of relations and some applications. In Shunro Watanabe and Morio Nagata, editors, *ISSAC '90: proceedings of the International Symposium on Symbolic and Algebraic Computation: August 20–24, 1990, Tokyo, Japan*, pages 24–29, New York, NY 10036, USA and Reading, MA, USA, 1990. ACM Press and Addison-Wesley.

[CDG+98] Hubert Comon, Max Dauchet, Remi Gilleron, Denis Lugiez, Sophie Tison, and Marc Tommasi. *Tree Automata Techniques and Applications.* Obtainable from `http://l3ux02.univ-lille3.fr/tata/`, 1998.

[CE81] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proc. IBM Workshop on Logics of Programs*, volume 131 of *LNCS*, pages 52–71. Springer-Verlag, 1981.

[CGZ96] Edmund M. Clarke, Steven M. German, and X. Zhao. Verifying the SRT division algorithm using theorem-proving techniques. In Alur and Henzinger [AH96], pages 111–122.

[Chv83] Vašek Chvátal. *Linear Programming*. W. H. Freeman and Company, New York, 1983.

[CK90] Chen Chung Chang and H. Jerome Keisler. *Model Theory*, volume 73 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, Amsterdam, 3rd edition, 1990. (1st ed., 1973; 2nd ed., 1977).

[CKM+91] Dan Craigen, Sentot Kromodimoeljo, Irwin Meisels, Bill Pase, and Mark Saaltink. EVES: An overview. In Soren Prehn and Hans Toetenel, editors, *Proceedings of Formal Software Development Methods (VDM '91)*, volume 552 of *LNCS*, pages 389–405, Berlin, Germany, October 1991. Springer.

[CLS96] David Cyrluk, Patrick Lincoln, and Natarajan Shankar. On Shostak's decision procedure for combinations of theories. In *Proc. of the* 13^{th} *Intl. Conference on Automated Deduction*, volume 1104 of *LNCS*, pages 463–477. Springer-Verlag, 1996.

[CMR97] David Cyrluk, Oliver Möller, and Harald Rueß. An efficient decision procedure for the theory of fixed-sized bit-vectors. In Grumberg [Gru97], pages 60–71.

[Col75] George E. Collins. *Quantifier Elimination for Real Closed Fields by Cylindrical Algebraic Decomposition, LNCS 32*. Springer Verlag, 1975.

[Col84] Alain Colmerauer. Equations and inequations on finite and infinite trees. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 85–99, 1984.

[Col93] Alain Colmerauer. Naive Solving of Non-linear Constraints. In Frédéric Benhamou and Alain Colmerauer, editors, *Constraint Logic Programming: Selected Research*, pages 89–112. MIT Press, 1993.

[Com90] Hubert Comon. Solving symbolic ordering constraints. *International Journal of Foundations of Computer Science*, 1(4):387–411, 1990.

[Coo72] D. C. Cooper. Theorem proving in arithmetic without multiplication. In *Machine Intelligence*, volume 7, pages 91–99. American Elsevier, 1972.

[CS70] John Cocke and Jack T. Schwartz. Programming languages and their compilers. Technical report, Courant Institute of Mathematical Sciences, New York, 1970.

[CU98] Michael A. Colón and Tomás E. Uribe. Generating finite-state abstractions of reactive systems using decision procedures. In Hu and Vardi [HV98], pages 293–304.

[Dam96] Dennis R. Dams. *Abstract Interpretation and Partition Refinement for Model Checking*. PhD thesis, Eindhoven University of Technology, July 1996.

[Dan62] George. B. Dantzig. *Linear Programming and Extensions*. Princeton Univ. Press, Princeton, NJ, 1962.

[Dav81] Martin Davis. Obvious logical inferences. In *Proceedings of the Seventh International Joint Conference on Artificial Intelligence*, pages 530–531, August 1981.

[Det96] Dave Detlefs. An overview of the extended static checking system. In *Proc. First Workshop on Formal Methods in Software Practice*, pages 1–9. ACM (SIGSOFT), January 1996.

[DGG94] Dennis R. Dams, Orna Grumberg, and Rob Gerth. Abstract interpretation of reactive systems: Abstractions preserving $\forall$CTL*, $\exists$ECTL*, CTL*. In *IFIP Working Conference on Programming Concepts, Methods and Calculi (PROCOMET 94)*, pages 573–592, June 1994.

[Diw98] Amer Diwan. Array bounds checking. In preparation, 1998.

[dK94] Eric de Kogel. Rigid E-Unification Simplified. In *TABLEAUX'94*, pages 17–30, 1994.

[DLL62] Martin Davis, George Logemann, and Donald W. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, July 1962.

[DP60] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *J. ACM*, 7:201–215, 1960.

[DS97] Andreas Dolzmann and Thomas Sturm. Simplification of quantifier-free formulae over ordered fields. *Journal of Symbolic Computation*, 24(2):209–231, August 1997.

[DST80] P. J. Downey, R. Sethi, and R. E. Tarjan. Variations on the common subexpression problem. *J. ACM*, 27(4):758–771, March 1980.

[DV95] Anatoli Degtyarev and Andrei Voronkov. Simultaneous rigid E-unification is undecidable. UPMAIL Technical Report No. 105, Computing Science Department, Uppsala University, 1995.

[DV96] Anatoli Degtyarev and Andrei Voronkov. What you always wanted to know about rigid E-Unification. In José Júlio Alferes, Luís Moniz Pereira, and Ewa Orlowska, editors, *Proceedings of the European JELIA Workshop (JELIA-96): Logics in Artificial Intelligence*, volume 1126 of *LNAI*, pages 50–69, Berlin, September 30–October3 1996. Springer.

[ELTT65] Y.L. Ershov, I.A. Lavrov, A.D. Taimanov, and M.A. Taitslin. Elementary theories. *Russ. Math. Survey*, 20:35–106, 1965.

[Fef96] Solomon Feferman. Computation on abstract data types. The extensional approach, with an application to streams. *Annals of Pure and Applied Logic*, 81:75–113, 1996.

[Fis98] Bernd Fischer. *Automatic software retrieval.* PhD thesis, TU Braunschweig, 1998.

[FMS98] Bernd Finkbeiner, Zohar Manna, and Henny B. Sipma. Deductive verification of modular systems. In Willem-Paul de Roever, Hans Langmaack, and Amir Pnueli, editors, *Compositionality: The Significant Difference, COMPOS'97*, volume 1536 of *LNCS*, pages 239–275. Springer-Verlag, 1998.

[Fri91] Alan M. Frisch. The substitutional framework for sorted deduction: Fundamental results on hybrid reasoning. *Artificial Intelligence*, 49:161–198, 1991.

[GD98] Shankar G. Govindaraju and David L. Dill. Verification by approximate forward and backward reachability. In *Proceedings of International Conference on Computer-Aided Design*, November 1998. San Jose, CA.

[GDHH98] Shankar G. Govindaraju, David L. Dill, Alan J. Hu, and Mark A. Horowitz. Approximate reachability with bdds using overlapping projections. In *Proceedings of the 35th Design Automation Conference*, June 1998. San Francisco, CA.

[Gen69] Gerhard Gentzen. Investigations into logical deductions, 1935. In M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pages 68–131. North-Holland Publishing Co., Amsterdam, 1969.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability - A Guide to the Theory of NP-Completeness*. Freeman, 1979.

[GNP+93] Jean Gallier, Paliath Narendran, David Plaisted, Stan Raatz, and Wayne Snyder. An algorithm for finding canonical sets of ground rewrite rules in polynomial time. *Journal of the ACM*, 40(1):1–16, January 1993.

[GNRS92] Jean Gallier, Paliath Narendran, Stan Raatz, and Wayne Snyder. Theorem proving using equational matings and rigid E-unification. *J. ACM*, 39(2):377–429, April 1992.

[Grä79] George Grätzer. *Universal Algebra*. Springer, New York, second edition, 1979.

[Gru97] Orna Grumberg, editor. *Proc.* 9^{th} *Intl. Conference on Computer Aided Verification*, volume 1254 of *LNCS*. Springer-Verlag, June 1997.

[GS97] Susanne Graf and Hassen Saidi. Construction of abstract state graphs with PVS. In Grumberg [Gru97], pages 72–83.

[Gut98] Claudio Gutierrez. Satisfiability of word equations with constants is in exponential space. In *FOCS'98*, 1998.

[GW75] Steven M. German and B. Wegbreit. A Synthesizer of Inductive Assertions. *IEEE transactions on Software Engineering*, 1(1):68–75, March 1975.

[HLL90] Tien Huynh, Catherine Lassez, and Jean-Louis Lassez. Fourier algorithm revisited. In Hélène Kirchner and Wolfgang Wechler, editors, *Algebraic and Logic Programming*, volume 463 of *LNCS*, pages 117–131. Springer-Verlag, 1990.

[HMD97] Pascal Van Hentenryck, Laurent Michel, and Yves Deville. *Numerica: A Modeling Language for Global Optimization*. The MIT Press, Cambridge, MA, USA, 1997.

[Hme76] Ju. I. Hmelevskiĭ. *Equations in Free Semigroups*. Number 107 in Proceedings of the Steklov Institute of Mathematics. American Mathematical Society, 1976. Translated from the Russian original: Trudy Mat. Inst. Steklov. 107, 1971.

[Hod93] Wilfred A. Hodges. *Model Theory*. Cambridge University Press, Cambridge, 1993.

[Hon92] Hoon Hong. Heuristic Search Strategies for Cylindrical Algebraic Decomposition. Technical Report 92-16, RISC-Linz, Johannes Kepler University, Linz, Austria, 1992.

[HSG98] Ravi Hosabettu, Mandayam K. Srivas, and Ganesh Gopalakrishnan. Decomposing the proof of correctness of pipelined microprocessors. In *Proc.* 10^{th} *Intl. Conference on Computer Aided Verification*, July 1998.

[HV95] J. N. Hooker and V. Vinay. Branching rules for satisfiability. *J. Automated Reasoning*, 15:359–383, 1995.

[HV98] Alan J. Hu and Moshe Y. Vardi, editors. *Proc.* 10^{th} *Intl. Conference on Computer Aided Verification*, volume 1427 of *LNCS*. Springer-Verlag, June 1998.

[IH93] Jean Louis Imbert and Pascal Van Hentenryck. Efficient Handling of Disequations in CLP over Linear Rational Arithmetics. In Frédéric Benhamou and Alain Colmerauer, editors, *Constraint Logic Programming: Selected Research*, pages 49–71. MIT Press, 1993.

[Iss90] Sunil Issar. Path-focused duplication: A search procedure for general matings. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 221–226, July–August 1990.

[Jaf81] Joxan Jaffar. Presburger arithmetic with array segments. *Information Processing Letters*, 12(2):79–82, April 1981.

[Jaf90] Joxan Jaffar. Minimal and complete word unification. *J. ACM*, 37(1):47–85, 1990.

[JK90] Jean Pierre Jouannaud and Claude Kirchner. Solving equations in abstract algebras: A rule based survey of unification. Technical report, University de Paris-Sud, 1990.

[JM94] Joxan Jaffar and M.J. Maher. Constraint logic programming: a survey. *Journal of Logic Programming*, 19-20(3):503–81, 1994.

[Kam96] Jeff Kamerer. Bus scheduler verification using STeP. Unpublished manuscript, Stanford University, Computer Science Department, January 1996.

[Kap97] Deepak Kapur. Shostak's congruence closure as completion. In *International Conference on Rewriting Techniques and Applications, RTA '97*, June 1997.

[Kla98] Nils Klarlund. MONA & FIDO: The logic-automaton connection in practice. In *Computer Science Logic, CSL '97*, LNCS, 1998.

[KM76] Shmuel Katz and Zohar Manna. Logical analysis of programs. *Communications of the ACM*, 19(4):188–206, April 1976.

[KMS98] Nils Klarlund, Anders Møller, and Michael Schwartzbach. QBDD representation using MONA. `http://www.brics.dk/mona/mona_examples.html`, 1998.

[Koz77] Dexter C. Kozen. Complexity of finitely presented algebras. In *Proc. 9th ACM Symp. Theory of Comp.*, pages 164–177, 1977.

[KP96] A. Koscielski and L. Pacholski. Complexity of Makanin's algorithms. *J. ACM*, 43(4), July 1996.

[KS97] Nils Klarlund and Michael Schwartzbach. A domain-specific language for regular sets of strings and trees. In *DSL'97*, 1997.

[LFMM92] Beth Levy, Ivan Filippenko, Leo Marcus, and Telis Menas. Using the state delta verification system (SDVS) for hardware verification. In Tom F Melham, V Stavridou, and Raymond T Boute, editors, *Theorem Provers in Circuit Design: Theory, Practice and Experience*, pages 337–360. Elsevier Science Publishers B.V. (North-Holland), Nijmegen, Netherlands, 1992.

[LGS+95] Claire Loiseaux, Susanne Graf, Joseph Sifakis, Ahmed Bouajjani, and Saddek Bensalem. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design*, 6:1–35, 1995.

[LHM93] Jean-Louis Lassez, Tien Huynh, and Ken McAloon. Simplification and elimination of redundant linear arithmetic constraints. In Frédéric Benhamou and Alain Colmerauer, editors, *Constraint Logic Programming: Selected Research*, pages 73–87. MIT Press, 1993.

[LO97] Jeremy Levitt and Kunle Olukotun. Verifying correct pipeline implementation for microprocessors. In *IEEE/ACM International Conference on Computer Aided Design; Digest of Technical Papers (ICCAD '97)*, pages 162–169, Washington - Brussels - Tokyo, November 1997. IEEE Computer Society Press.

[LW93] Rüdiger Loos and Volker Weispfenning. Applying linear quantifier elimination. *The Computer Journal*, 36(5):450–462, October 1993.

[MAB+94] Zohar Manna, Anuchit Anuchitanukul, Nikolaj Bjørner, Anca Browne, Eddie S. Chang, M. Colón, Luca de Alfaro, Harish Devarajan, Henny B. Sipma, and Tomás E. Uribe. STeP: The Stanford temporal prover. Technical Report STAN-CS-TR-94-1518, Computer Science Department, Stanford University, July 1994.

[Mah88a] Michael J. Maher. Complete axiomatizations of the algebras of finite, rational and infinite trees. In *Proc. 3rd IEEE Symp. Logic in Comp. Sci.*, pages 348–357, 1988.

[Mah88b] Michael J. Maher. Complete axiomatizations of the algebras of finite, rational and infinite trees. Technical report, IBM Research Report, T.J. Watson Research Center, 1988.

[Mak77] G. S. Makanin. The problem of solvability of equations in a free semigroup. *Math. USSR Sbornik*, 32(2):129–198, 1977.

[Mal71] Anatoli Mal'cev. *The Metamathematics of Algebraic Systems.* North-Holland, 1971.

[Mat81] Prabhaker Mateti. A decision procedure for the correctness of a class of programs. *Journal of the ACM*, 28(2):215–222, April 1981.

[McA91] David McAllester. Grammar rewriting. Technical Report A.I. Memo No. 1342, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, December 1991.

[McM93] Ken L. McMillan. *Symbolic Model Checking.* Kluwer Academic Pub., 1993.

[Meg83] Nimrod Megiddo. Towards a genuinely polynomial algorithm for linear programming. *SIAM Journal on Computing*, 12(2):347–353, 1983.

[Mil89] Robin Milner. *Communication and Concurrency.* Prentice Hall, New York, 1989.

[Min92] Grigory Mints. *Selected papers in proof theory.* Napoli, Bibliopolis, 1992.

[MM82] Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Trans. Prog. Lang. Sys.*, 4(2):258–282, 1982.

[MNT98] Martin Müller, Joachim Niehren, and Ralf Treinen. The first-order theory of ordering constraints over feature trees. In *Thirteenth annual IEEE Symposium on Logic in Computer Sience (LICS98)*, Indianapolis, Indiana, 21-24 June 1998.

[Mos88] Louise E. Moser. A decision procedure for unquantified formulas of graph theory. In E. Lusk; R. Overbeek, editor, *Proceedings on the 9th International Conference on Automated Deduction*, volume 310 of *LNCS*, pages 344–357, Berlin, May 1988. Springer.

[MP94] Zohar Manna and Amir Pnueli. Temporal verification diagrams. In M. Hagiya and John C. Mitchell, editors, *Proc. International Symposium on Theoretical Aspects of Computer Software*, volume 789 of *LNCS*, pages 726–765. Springer-Verlag, 1994.

[MP95] Zohar Manna and Amir Pnueli. *Temporal Verification of Reactive Systems: Safety.* Springer-Verlag, New York, 1995.

[MR98] Oliver Möller and Harald Rueß. Solving bit-vector equalities. In *Formal Methods in Computer Aided Design*, volume 1522 of *LNCS*, pages 36–48. Springer Verlag, 1998.

[MS98] Zohar Manna and Henny B. Sipma. Deductive verification of hybrid systems using STeP. In T.A. Henzinger and S. Sastry, editors, *Hybrid Systems: Computation and Control*, volume 1386 of *LNCS*, pages 305–318. Springer-Verlag, 1998.

[MSW91] Zohar Manna, Mark E. Stickel, and Richard Waldinger. Monotonicity properties in automated deduction. In V. Lifschitz, editor, *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, pages 261–280. Academic Press, Boston, MA, 1991.

[MW86] Zohar Manna and Richard Waldinger. Special relations in automated deduction. *J. ACM*, 33(1):1–59, 1986.

[MW92] Zohar Manna and Richard Waldinger. The special-relation rules are incomplete. In *Proc. of the* 11^{th} *Intl. Conference on Automated Deduction*, volume 607 of *LNCS*, pages 492–506. Springer-Verlag, 1992.

[MW93] Zohar Manna and Richard Waldinger. *The Deductive Foundations of Computer Programming*. Addison-Wesley, Reading, MA, 1993.

[Nel81] Greg C. Nelson. Techniques for Program Verification. Technical Report CSL-81-10, Xerox Palo Alto Research Center Research Report, 1981.

[Nel82] Greg C. Nelson. An $n^{\log n}$ algorithm for the two-variable-per-constraint linear programming satisfiability problem. Technical report, Stanford University, 1982.

[NG98] Ratan Nalumasu and Ganesh Gopalakrishnan. Deriving efficient cache coherence protocols through refinement. In *Third International Workshop on Formal Methods for Parallel Programs: Theory and Applications '98*, April 1998.

[NO78] Greg Nelson and Derek C. Oppen. Fast decision procedures based on congruence closure. Technical Report STAN-CS-77-646, Computer Science Department, Stanford University, Stanford, California, February 1978.

[NO79] Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. *ACM Trans. Prog. Lang. Sys.*, 1(2):245–257, October 1979.

[NO80] Greg Nelson and Derek C. Oppen. Fast decision procedures based on congruence closure. *J. ACM*, 27(2):356–364, April 1980.

[NP93] J. Niehren and A. Podelski. Feature automata and recognizable sets of feature trees. In Marie-Claude Gaudel and Jean-Pierre Jouannaud, editors, *TAPSOFT 93: Theory and Practice of Software Development*, volume 668 of *LNCS*, pages 356–375. Springer-Verlag, April 1993.

[NZM91] I. Niven, H.S. Zuckerman, and H.L. Montgomery. *An Introduction to the Theory of Numbers*. John Wiley & Sons, New York, 1991.

[Opp80a] Derek C. Oppen. Complexity, convexity and combinations of theories. *Theoretical Computer Science*, 12(3):291–302, November 1980.

[Opp80b] Derek C. Oppen. Reasoning about recursively defined data structures. *J. ACM*, 27(3), July 1980.

[ORR+96] Sam Owre, Sreeranga Rajan, John M. Rushby, Natarajan Shankar, and Mandayam K. Srivas. PVS: Combining specification, proof checking, and model checking. In Alur and Henzinger [AH96], pages 411–414.

[Pau93] Lawrence C. Paulson. Co-induction and Co-recursion in Higher-order Logic. Technical report, University of Cambridge, 1993.

[Pau97] Lawrence C. Paulson. Mechanizing coinduction and corecursion in higher-order logic. *Journal of Logic and Computation*, 7(2):175–204, April 1997.

[Plo72] Gordon Plotkin. Building in equational theories. *Machine Intelligence*, 7:73–90, 1972.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *Proc. 18th IEEE Symp. Found. of Comp. Sci.*, pages 46–57. IEEE Computer Society Press, 1977.

[Pra77] Vaughan Pratt. Two simple theories whose combination is hard. Unpublished, 1977.

[Pro93] Patrick Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9(3):268–299, August 1993. (Also available as Technical Report AISL-46-91, Stratchclyde, 1991).

[PT87] Robert Paige and Robert E. Tarjan. Three partition refinement algorithms. *SIAM Journal on Computing*, 16(6):973–989, 1987.

[Pug91] William Pugh. The Omega test: a fast and practical integer programming algorithm for dependence analysis. In IEEE, editor, *Proceedings, Supercomputing '91: Albuquerque, New Mexico, November 18–22, 1991*, pages 4–13, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1991. IEEE Computer Society Press.

[Qui46] William V. O. Quine. Concatenations as basis for arithmetic. *J. Symb. Logic*, 11:105–119, 1946.

[Rac75] C. Rackoff. The computational complexity of some logical theories. Technical Report MIT-LCS//MIT/LCS/TR-144, Massachusetts Institute of Technology, Laboratory for Computer Science, February 1975.

[Rob65] J.A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, 1965.

[RP89] P. Ruzick and I. Privara. An almost linear Robinson unification algorithm. *Acta Informatica*, 27:61–71, 1989.

[RSS96] Harald Rueß, Natarajan Shankar, and Mandayam K. Srivas. Modular verification of SRT division. *Lecture Notes in Computer Science*, 1102:123–134, 1996.

[Sch86] Alexander Schrijver. *Theory of Linear and Integer Programming.* John Wiley & Sons, New York, 1986.

[Sch93] Klaus U. Schulz. Word unification and transformation of generalized equations. *Journal of Automated Reasoning*, 11(2):149–184, October 1993.

[SDB96] Jeffrey X. Su, David L. Dill, and Clark W. Barrett. Automatic generation of invariants for processor verification. In *1st Intl. Conf. on Formal Methods in Computer-Aided Design*, volume 1166 of *LNCS*, pages 377–388. Springer-Verlag, November 1996.

[Sha93] Natarajan Shankar. A lazy approach to compositional verification. Technical report, Computer Science Laboratory, SRI International, Menlo Park, California, December 1993.

[Sho77] Robert E. Shostak. On the SUP-INF method for proving Presburger formulas. *J. ACM*, 24(4):529–543, 1977.

[Sho78] Robert E. Shostak. An algorithm for reasoning about equality. *Communications of the ACM*, 21(2):583–585, July 1978.

[Sho79] Robert E. Shostak. A practical decision procedure for arithmetic with function symbols. *J. ACM*, 26(2):351–360, April 1979.

[Sho81] Robert E. Shostak. Deciding linear inequalities by computing loop residues. *J. ACM*, 28(4):769–779, October 1981.

[Sho84] Robert E. Shostak. Deciding combinations of theories. *J. ACM*, 31(1):1–12, January 1984.

[Sip98] Henny B. Sipma. *Diagram-based Verification of Discrete, Real-time and Hybrid Systems.* PhD thesis, Computer Science Department, Stanford University, December 1998.

[SJ80] N. Suzuki and D. Jefferson. Verification of Presburger array programs. *Jrnl. A.C.M.*, 27(1):191–205, January 1980.

[SM73] Larry J. Stockmeyer and Albert R. Meyer. Word problems requiring exponential time. In *Proc. 5rd ACM Symp. Theory of Comp.*, pages 1–9, 1973.

[Smo92] Gert Smolka. Feature constraint logics for unification grammars. *Journal of Logic Programming*, 12:51–87, 1992.

[SS98] Mary Sheeran and Gunnar Stålmarck. A tutorial on stålmarck's proof procedure for propositional logic. In *Formal methods in Computer Aided Verification*, volume 1522 of *LNCS*, pages 82–99, November 1998.

[Sti85] Mark E. Stickel. Automated deduction by theory resolution. *J. Automated Reasoning*, 1(4):333–355, 1985.

[SUM96] Henny B. Sipma, Tomás E. Uribe, and Zohar Manna. Deductive model checking. In Alur and Henzinger [AH96], pages 208–219.

[SV94] Bolek K. Szymanski and J. M. Vidal. Automatic verification of a class of symmetric parallel programs. In *Proc. 13th IFIP World Computer Congress*, volume A-51, pages 571–576, 1994.

[SvH95] Vijay Saraswat and Pascal van Hentenryck, editors. *Principles and Practice of Constraint Programming.*, Cambridge, MA, 1995. MIT Press.

[Tar51] Alfred Tarski. *A Decision Method for Elementary Algebra and Geometry.* University of California Press, Berkeley and Los Angeles, 1951.

[Tar75] Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. *J. ACM*, 22(2):215–225, April 1975.

[TH96] Cesare Tinelli and Mehdi Harandi. A new correctness proof of the Nelson–Oppen combination procedure. In Baader and Schulz [BS96], pages 103–120.

[TR98] Cesare Tinelli and Christophe Ringeissen. Combination of non-disjoint theories. First results. Technical report, University of Illinois at Urbana-Champaign, April 1998.

[Tre91] Ralf Treinen. First-Order Data Types and First-Order Logic. In Takayasu Ito and Albert R. Meyer, editors, *Theoretical Aspects of Computer Software*, Lecture Notes in Computer Science, vol. 526, pages 594–614, Sendai, Japan, 24–27 September 1991. Springer-Verlag.

[Tre92] Ralf Treinen. A New Method for Undecidability Proofs of First-Order Theories. *Journal of Symbolic Computation*, 14(5):437–457, November 1992.

[Tre96] Ralf Treinen. Feature trees over arbitrary structures. In Patrick Blackburn and Maarten de Rijke, editors, *Specifying Syntactic Structures*, Studies in Logic, Language and Information. CSLI Publications, 1996.

[Tul94] Sauro Tulipani. Decidability of the existential theory of infinite terms with subterm relation. *Information and Computation*, 108(1):1–33, January 1994.

[Uri98] Tomás E. Uribe. *Abstraction-based Deductive-Algorithmic Verification of Reactive Systems*. PhD thesis, Computer Science Department, Stanford University, December 1998.

[Ven87] K. N. Venkataraman. Decidability of the purely existential fragment of the theory of term algebras. *J. ACM*, 34(2):492–510, April 1987.

[Vor96] Sergei Vorobyov. An Improved Lower Bound for the Elementary Theories of Trees. In *Proc. of the* 13^{th} *Intl. Conference on Automated Deduction*, volume 1104 of *LNCS*, April 1996.

[WD95] Howard Wong-Toi and David L. Dill. Verification of real-time systems by successive over and under approximation. In *International Conference on Computer-Aided Verification*, Lecture Notes in Computer Science. Springer-Verlag, 1995.

[Wei94] Volker Weispfenning. Quantifier elimination for real algebra-the cubic case. In ACM, editor, *ISSAC '94: Proceedings of the 1994 International Symposium on Symbolic and Algebraic Computation: July 20–22, 1994, Oxford, England, United Kingdom*, pages 258–263, New York, NY 10036, USA, 1994. ACM Press.

[Wei97] Volker Weispfenning. Quantifier elimination for real algebra—the quadratic case and beyond. In *Applied Algebra and Error-Correcting Codes (AAECC) 8*, pages 85–101, 1997.

[Zha92] Hantao Zhang. A linear Robinson unification algorithm. Technical report, Argonne, Illinois., 1992. Presented at U.S./Japan Joint Workshop on Automated Reasoning.

Index